



FORNAX

Introduction to Normalizing Flows

Krzysztof Kolasiński, 20.09.2018

WWW.FORNAX.AI

Content

Part 1. Introduction

- Motivations for probabilistic models
- Theory of Normalizing Flows + simple examples
- Using Normalizing Flows in Tensorflow

Part 2. Research papers study

- MADE, IAF, MAF

Disclaimer

1. possible lack of knowledge in probabilistic modeling
2. possible errors in the equations,
3. possible errors in naming stuff
4. sorry for random oversimplification of the notation
5. rare changes in notation (the Copy-Paste effect)
6. This is the first part, which will be more fundamental

Part 1. Probabilistic modeling motivations and background

Motivation: The problem of density estimation

Consider following problem:

- we observe some data \mathbf{X} which are sample from some distribution $p(\mathbf{X})$
- we want to tell what is $p(\mathbf{X}')$ where \mathbf{X}' is some new data.

Example: anomaly detection

The simplest classical approach for this problem would be:

Take the data

```
array([[ 1.35791411,  0.41725294],  
       [-0.46717576,  0.72275205],  
       [ 1.01548965,  0.65188145],  
       [ 0.89550028, -0.41804664],  
       [ 0.90974391,  0.40584828],  
       [-0.59003111,  0.92042647],  
       [-1.52335804, -0.91684024],  
       [ 0.25105829, -0.66614504],  
       [-0.77281828, -1.28366714],  
       [-0.4297637 , -0.61117532]])
```

Fit e.g. multivariate gaussian

$$p_{\text{gaussian}}(\mathbf{X})$$

Find anomalies:

$$p_{\text{gaussian}}(\mathbf{X}') < \varepsilon$$

Motivation: The problem of sampling

Consider following problem:

- we observe some data \mathbf{X} which are sample from some distribution $p(\mathbf{X})$
- we want to generate new data point \mathbf{X}' according to $p(\mathbf{X})$

Example: dataset augmentation, latent space manipulation, generating new samples

The current state of the art approach for this problem would be to take some deep generative neural network e.g. Glow:



Example from: <https://blog.openai.com/glow/>

Motivation: probabilistic modeling

- Probabilistic models have nice theoretical interpretation
- However they are usually hard to train or they have intractable normalizing constants

Typical probabilistic model is specified as some function

$$q(\mathbf{x}; \theta) = \frac{f(\mathbf{x}; \theta)}{Z_\theta}$$

where:

- $f(\mathbf{x}; \theta)$ is some positive function parameterized with theta, usually neural network
- Z is normalization constant, contains sum of $f(\mathbf{x})$ over all possible configurations of \mathbf{x}

Unfortunately:

- in many cases Z is intractable, inference is impossible or require approximations
- sampling may be hard, so we cannot generate samples easily

Normalizing Flows: try to overcome this problem by introducing continuous transformation of probability density which allows for tracking the change in the density $p(\mathbf{x})$

Density estimation:
maximum likelihood estimation (MLE)

$$p_{\text{model}}(\mathbf{X})$$

Recap: Maximum likelihood estimation

Assumptions and definitions:

- probability of observing data point \mathbf{x} (unknown): $P_{\text{data}}(\mathbf{x})$
- samples drawn from p_{data} are independent (like images, CIFAR, MNIST)

$$P_{\text{data}}(\mathbf{X}) = P_{\text{data}}(\mathbf{x}_1) P_{\text{data}}(\mathbf{x}_1) \dots P_{\text{data}}(\mathbf{x}_N) = \prod_i P_{\text{data}}(\mathbf{x}_i)$$

- we want to create a model from which we can sample: $P_{\text{model}}(\mathbf{x})$
- in order to build as much “realistic” model as we can we want to have: $P_{\text{model}}(\mathbf{x}) = P_{\text{data}}(\mathbf{x})$
- this allows us to define target metric which tells us how far from true distribution we are:

$$D(P_{\text{data}}(\mathbf{x}), P_{\text{model}}(\mathbf{x})) = 0 \text{ if } P_{\text{model}}(\mathbf{x}) = P_{\text{data}}(\mathbf{x})$$



Recap: Maximum likelihood estimation

Assumptions and definitions:

- The **natural choice** for **D** is Kullback Leibler divergence (*this is a critical point for all derivations below*):

$$\text{KL} (P_{\text{data}} (\mathbf{x}) , P_{\text{model}} (\mathbf{x})) = \sum_i P_{\text{data}} (\mathbf{x}) \log \left(\frac{P_{\text{data}} (\mathbf{x})}{P_{\text{model}} (\mathbf{x})} \right)$$

- Our generator P_{model} will be defined by some neural network, let's put explicitly its dependence on some parameters (neural network weights)

$$P_{\text{model}} (\mathbf{x}) = P_{\text{model}} (\mathbf{x}; \theta)$$

- We want to minimize KL w.r.t model parameters, hence we need to compute the gradients

$$\nabla_{\theta} \text{KL} = \nabla_{\theta} \sum_i P_{\text{data}} (\mathbf{x}) \log \left(\frac{P_{\text{data}} (\mathbf{x})}{P_{\text{model}} (\mathbf{x}; \theta)} \right) = -\nabla_{\theta} \sum_i P_{\text{data}} (\mathbf{x}) \log (P_{\text{model}} (\mathbf{x}; \theta))$$

- The samples are explicitly sampled from data distribution (N goes to infinity):

$$\nabla_{\theta} \text{KL} = -\frac{1}{N} \nabla_{\theta} \sum_{\mathbf{x}_i \sim P_{\text{data}}(\mathbf{x})} \log (P_{\text{model}} (\mathbf{x}_i; \theta)) = -\frac{1}{N} \nabla_{\theta} \log \prod_{\mathbf{x}_i \sim P_{\text{data}}(\mathbf{x})} P_{\text{model}} (\mathbf{x}_i; \theta)$$

Recap: Maximum likelihood estimation

- The samples are explicitly sampled from data distribution (N goes to infinity):

$$\nabla_{\theta} \text{KL} = -\frac{1}{N} \nabla_{\theta} \sum_{\mathbf{x}_i \sim P_{\text{data}}(\mathbf{x})} \log (P_{\text{model}}(\mathbf{x}_i; \theta)) = -\frac{1}{N} \nabla_{\theta} \log \prod_{\mathbf{x}_i \sim P_{\text{data}}(\mathbf{x})} P_{\text{model}}(\mathbf{x}_i; \theta)$$

- **Recall assumption that:** $P_{\text{data}}(\mathbf{X}) = P_{\text{data}}(\mathbf{x}_1) P_{\text{data}}(\mathbf{x}_1) \dots P_{\text{data}}(\mathbf{x}_N) = \prod_i P_{\text{data}}(\mathbf{x}_i)$
- This allows us to write similar expression for model (but X are drawn from “real” data)

$$\nabla_{\theta} \text{KL} = -\frac{1}{N} \nabla_{\theta} \log P_{\text{model}}(\mathbf{X}; \theta)$$

- Hence by minimizing **KL divergence** we **maximize log-likelihood** of observed data, both can be used to obtain same result. However with one it maybe do in easier way.

minimize KL maximize $\log P_{\text{model}}(\mathbf{X}; \theta)$

Recap: MLE for simple gaussian problem - a textbook example

Consider following problem:

- we observe some data \mathbf{X} which are sample from some distribution $p(\mathbf{X})$
- we want to tell what is $p(\mathbf{X}')$ where \mathbf{X}' is some new data.

Take the data:

```
array([[ 1.35791411,  0.41725294],  
       [-0.46717576,  0.72275205],  
       [ 1.01548965,  0.65188145],  
       [ 0.89550028, -0.41804664],  
       [ 0.90974391,  0.40584828],  
       [-0.59003111,  0.92042647],  
       [-1.52335804, -0.91684024],  
       [ 0.25105829, -0.66614504],  
       [-0.77281828, -1.28366714],  
       [-0.4297637 , -0.61117532]])
```

Build a model:

$$p_{\text{model}}(x; \theta) = \frac{1}{\sqrt{2\pi}\sigma} \exp \left[-\frac{(x - \mu)^2}{2\sigma^2} \right]$$

$$\theta = \{\mu, \sigma\}$$

Define likelihood function - the cost

$$\log p_{\text{model}}(X; \theta) = \log \prod_i p_{\text{model}}(x_i; \theta)$$

Find the optimal model parameters by solving problem:

$$\theta^* = \underset{\theta}{\operatorname{argmax}} \log p_{\text{model}}(X; \theta)$$

Recap: MLE for simple gaussian problem - a textbook example

$$\theta^* = \operatorname{argmax}_{\theta} \log p_{\text{model}}(X; \theta)$$

$$p_{\text{model}}(x; \theta) = \frac{1}{\sqrt{2\pi}\sigma} \exp \left[-\frac{(x - \mu)^2}{2\sigma^2} \right]$$

Once the problem is solved we can:

- estimate $p_{\text{model}}(x')$
- sample from p_{model} , with following sampling rule $x = \mu + \varepsilon\sigma$ with $\varepsilon \sim \mathcal{N}(0, 1)$

Probabilistic change of variables a.k.a
Normalizing Flows

Normalizing flows - simplest case - linear transform in 1D

Normalizing Flows allow for defining complex densities by transforming simple one by invertible mappings i.e. bijections.

Let's build a simple model for the simple example considered in the previous slide.

- we start from the simplest univariate gaussian distribution

$$\varepsilon \sim \mathcal{N}(0, 1)$$

- we define a affine transformation (a forward pass)

$$x = \mu + \varepsilon\sigma$$

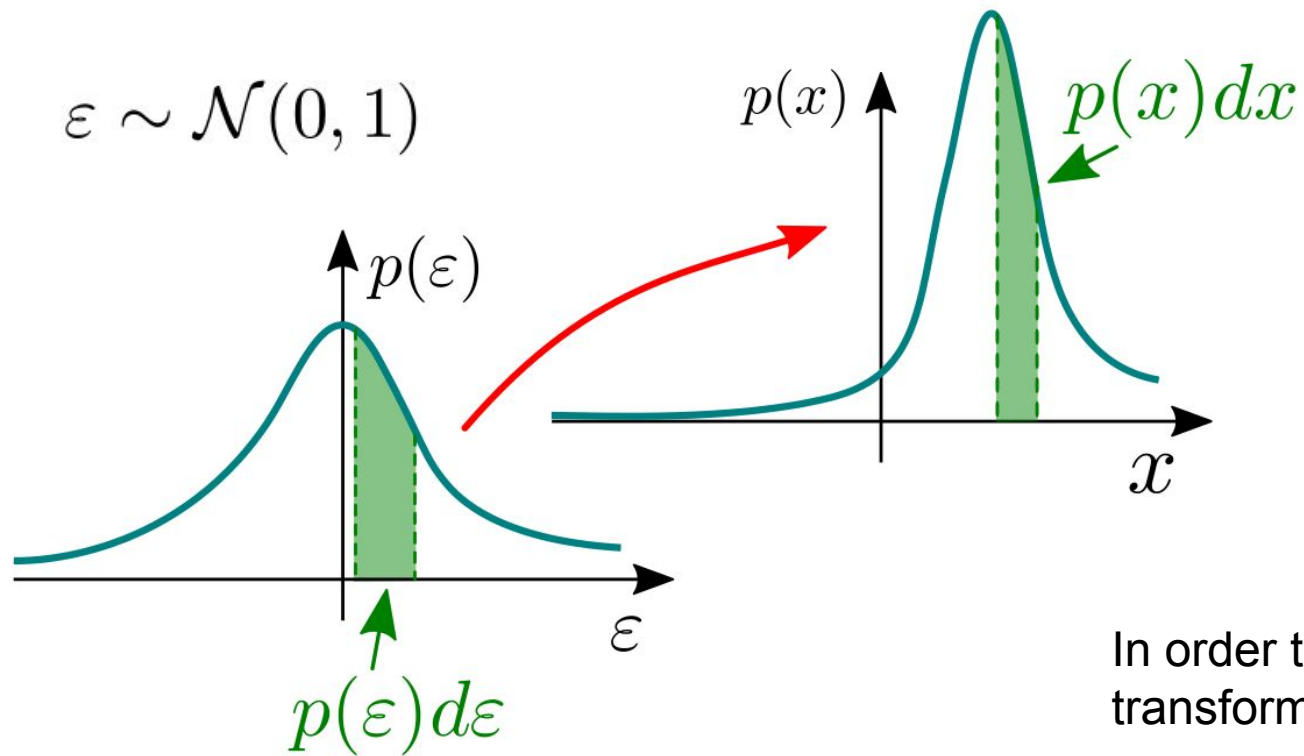
- and inverse (with constraint $\sigma > 0$):

$$\varepsilon = (x - \mu) / \sigma$$

- we know $p(\varepsilon)$ and we want to find $p(x)$

Normalizing flows - transforming scalars

The volume change, but density must be preserved:



The necessary condition for this is:

$$p(\varepsilon)d\varepsilon = p(x)dx$$

The transformed density is then:

$$x = \mu + \varepsilon\sigma$$
$$p(x) = p(\varepsilon) \left| \frac{dx}{d\varepsilon} \right|^{-1} = \frac{p(\varepsilon)}{\sigma}$$

change of volume

In order to estimate the density at x we have to apply inverse transform:

$$\varepsilon = (x - \mu) / \sigma$$

Which will result in:

$$p_{\text{model}}(x; \theta) = \frac{1}{\sqrt{2\pi}\sigma} \exp \left[-\frac{(x - \mu)^2}{2\sigma^2} \right]$$

Normalizing flows - transforming scalars

Let us now consider a slightly more complicated example. Note the change of notation.

$$z^{(0)} \sim p^{(0)}(z)$$

$$z^{(1)} = f^{(1)}(z^{(0)})$$

$$z^{(2)} = f^{(2)}(z^{(1)})$$

In some cases we are interested in computing a probability density of sampled data point.
This requires a **forward** propagation of the flow.

$$p(x) = p(\varepsilon) \left| \frac{dx}{d\varepsilon} \right|^{-1}$$

transformation equation

Density transformation after first map

$$p^{(1)}(z^{(1)}) = p^{(0)}(z^{(0)}) \left| \frac{dz^{(1)}}{dz^{(0)}} \right|^{-1}$$

Density transformation after second map

$$p^{(2)}(z^{(2)}) = p^{(1)}(z^{(1)}) \left| \frac{dz^{(2)}}{dz^{(1)}} \right|^{-1}$$

Density of the sample:

$$p^{(2)}(z^{(2)}) = p^{(0)}(z^{(0)}) \left| \frac{dz^{(1)}}{dz^{(0)}} \right|^{-1} \left| \frac{dz^{(2)}}{dz^{(1)}} \right|^{-1}$$

This is called a forward pass

Normalizing flows - transforming scalars: inverse transform

Sampling:

$$z^{(0)} \sim p^{(0)}(z)$$

Density of the sample

$$z^{(1)} = f^{(1)}(z^{(0)})$$

$$z^{(2)} = f^{(2)}(z^{(1)})$$

$$p^{(2)}(z^{(2)}) = p^{(0)}(z^{(0)}) \left| \frac{dz^{(1)}}{dz^{(0)}} \right|^{-1} \left| \frac{dz^{(2)}}{dz^{(1)}} \right|^{-1}$$

In other cases we are interested in estimating density of external samples. A computation of the **inverse** process is required.

We start with some external sample \tilde{z} and build an inverse flow

$$\tilde{z}^{(1)} = g^{(2)}(\tilde{z})$$

$$\tilde{z}^{(0)} = g^{(1)}(\tilde{z}^{(1)})$$

g is an inverse map

$$g = f^{-1}$$

From analogy to forward flow we can write formula for **inverse flow**:

$$p(\tilde{z}) = \left| \frac{d\tilde{z}^{(1)}}{d\tilde{z}} \right| \left| \frac{d\tilde{z}^{(0)}}{d\tilde{z}^{(1)}} \right| p^{(0)}(\tilde{z}^{(0)})$$

Recal simple example:

$$p(x) = p(\varepsilon) \left| \frac{dx}{d\varepsilon} \right|^{-1} = \frac{p(\varepsilon)}{\sigma}$$

Normalizing flows - forward and inverse flows summary

Forward flow:

$$z^{(0)} \sim p^{(0)}(z)$$

Density of the sample

$$z^{(1)} = f^{(1)}(z^{(0)})$$

$$p^{(2)}(z^{(2)}) = p^{(0)}(z^{(0)}) \left| \frac{dz^{(1)}}{dz^{(0)}} \right|^{-1} \left| \frac{dz^{(2)}}{dz^{(1)}} \right|^{-1}$$

$$z^{(2)} = f^{(2)}(z^{(1)})$$

Used for sampling, useful in the context of **Variational autoencoders**.

Inverse flow:

$$\tilde{z}^{(1)} = g^{(2)}(\tilde{z})$$

$$\tilde{z}^{(0)} = g^{(1)}(\tilde{z}^{(1)})$$

$$p(\tilde{z}) = \left| \frac{d\tilde{z}^{(1)}}{d\tilde{z}} \right| \left| \frac{d\tilde{z}^{(0)}}{d\tilde{z}^{(1)}} \right| p^{(0)}(\tilde{z}^{(0)})$$

Density of the external data point

Can be used for anomaly detection, and with MLE.

Normalizing flows - generalization

- So far we worked with scalars, but NFs can be extended to multidimensional problems.
- Let \mathbf{z} be a random vector with probability density function $q(\mathbf{z})$ and $f(\mathbf{z})$ is an invertible function.
- Transformation $\mathbf{z}^{(n+1)} = f(\mathbf{z}^{(n)})$ of the random variable leads to the following change in the probability density:

$$q'(\mathbf{z}') = q(\mathbf{z}) \left| \det \frac{\partial f(\mathbf{z}^{(n)})}{\partial \mathbf{z}^{(n)}} \right|^{-1}$$

det = determinant **jacobian matrix**

$$\frac{\partial f(\mathbf{z})}{\partial \mathbf{z}} = \begin{pmatrix} \frac{\partial f_1}{\partial z_1} & \dots & \frac{\partial f_1}{\partial z_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_n}{\partial z_1} & \dots & \frac{\partial f_n}{\partial z_n} \end{pmatrix}$$

Normalizing flows - generalization

- Stacking multiple mappings leads to final expression for the normalizing flow - a chain rule

$$\begin{aligned} \mathbf{z}^{(0)} &\sim p(\mathbf{z}) \\ \mathbf{z}^{(1)} &= f^{(1)}(\mathbf{z}^{(0)}) \\ \mathbf{z}^{(2)} &= f^{(2)}(\mathbf{z}^{(1)}) \\ &\dots \\ \mathbf{z}^{(n)} &= f^{(n)}(\mathbf{z}^{(n-1)}) \end{aligned} \quad \longrightarrow \quad q^{(n)}(\mathbf{z}^{(n)}) = q(\mathbf{z}^{(0)}) \prod_{i=1}^n \left| \det \frac{\partial \mathbf{z}^{(i)}}{\partial \mathbf{z}^{(i-1)}} \right|^{-1}$$

density of the sampled point

- Note, that in practice due to the numerical errors it is more stable to work with logarithms

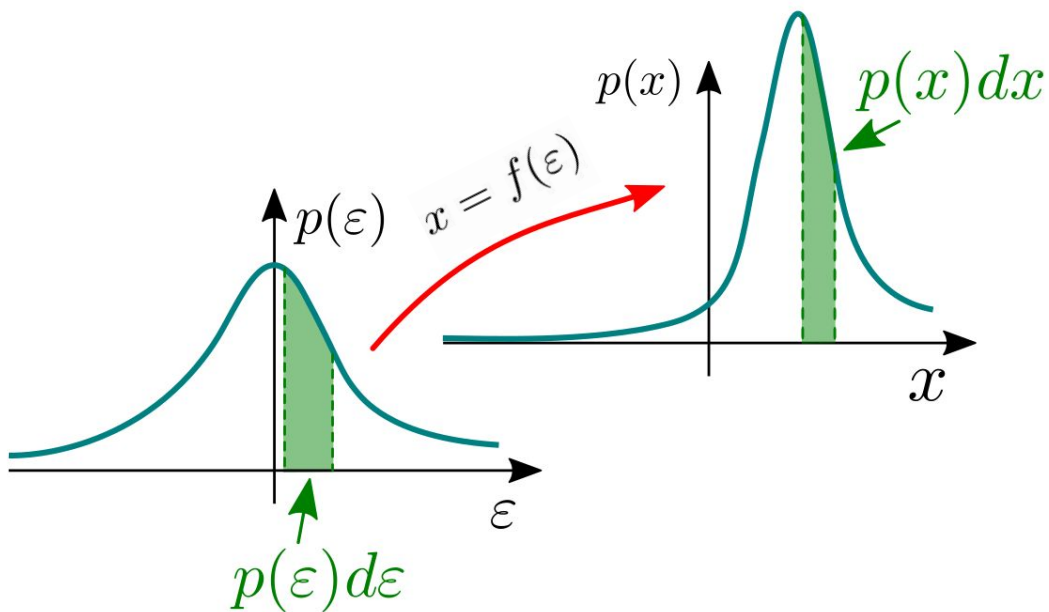
$$\log q^{(n)}(\mathbf{z}^{(n)}) = \log q(\mathbf{z}^{(0)}) - \sum_{i=1}^n \log \left| \det \frac{\partial \mathbf{z}^{(i)}}{\partial \mathbf{z}^{(i-1)}} \right|$$

Normalizing flows - determinant

- The final formula for NFs contains expression which depends on the determinant of the jacobian transformation.

$$\log q^{(n)}(\mathbf{z}^{(n)}) = \log q(\mathbf{z}^{(0)}) - \sum_{i=1}^n \log \left| \det \frac{\partial \mathbf{z}^{(i)}}{\partial \mathbf{z}^{(i-1)}} \right|$$

- Lets get some intuition how to interpret it.
- In 1D we got:



Mass conservation formula: $p(\varepsilon)d\varepsilon = p(x)dx$

In general we should talk about volumes, which are scalars (since we work with densities)

$$p(\varepsilon)dV_\varepsilon = p(x)dV_x$$

Source space is cartesian, so the volume is $dV_\varepsilon = d\varepsilon$

$$p(x) = p(\varepsilon) \left| \frac{dV_x}{dV_\varepsilon} \right|^{-1}$$

$$q'(\mathbf{z}') = q(\mathbf{z}) \left| \det \frac{\partial f(\mathbf{z}^{(n)})}{\partial \mathbf{z}^{(n)}} \right|^{-1}$$

determinant here is related to the change in the volume affected by the transformation.

Normalizing flows - determinant - 2D case

- We are going to show the presence of the determinant on very simple 2D example, where we will show geometric interpretation of the determinant.

- Consider following an invertible mapping f

$$y_1 = f_1(x_1, x_2) \quad \mathbf{y} = \mathbf{f}(\mathbf{x})$$

$$y_2 = f_2(x_1, x_2) \quad \text{a vectorized expression}$$

we want to compute this

$$p(\mathbf{y}) = p(\mathbf{x}) \left| \frac{dV_{\mathbf{y}}}{dV_{\mathbf{x}}} \right|^{-1}$$

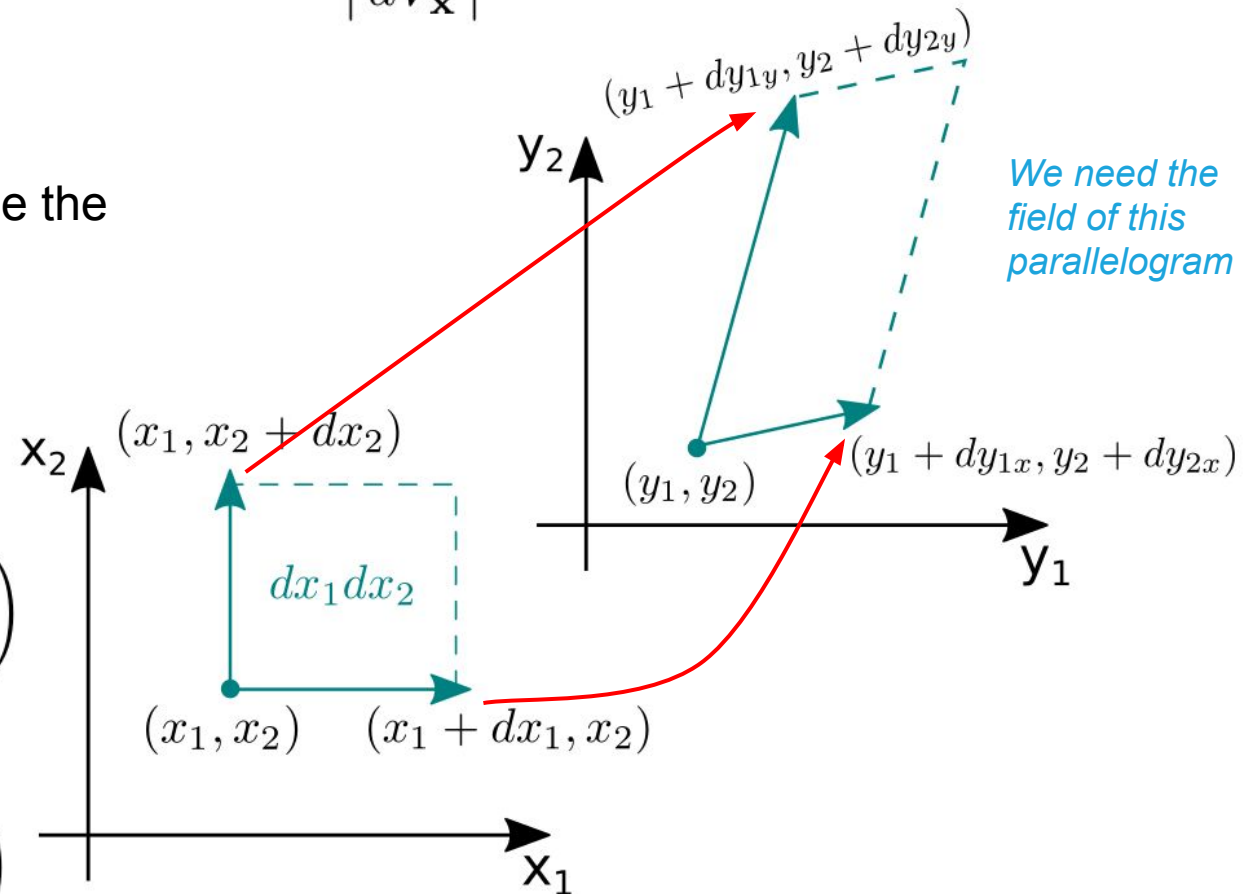
- An infinitesimal change in the x coordinate will cause the change in the transformed space for both z_1 and z_2 :

$$\begin{cases} f_1(x_1 + dx_1, x_2) = f_1(x_1, x_2) + \frac{\partial f_1}{\partial x_1} dx_1 \\ f_2(x_1 + dx_1, x_2) = f_2(x_1, x_2) + \frac{\partial f_2}{\partial x_1} dx_1 \end{cases}$$

$$(x_1 + dx_1, x_2) \rightarrow \left(y_1 + \frac{\partial f_1}{\partial x_1} dx_1, y_2 + \frac{\partial f_2}{\partial x_1} dx_1 \right)$$

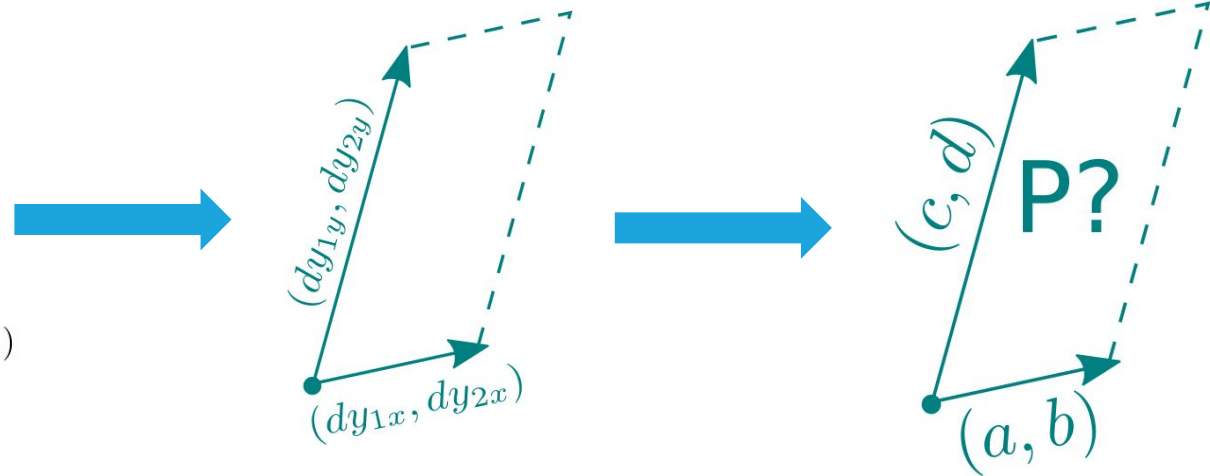
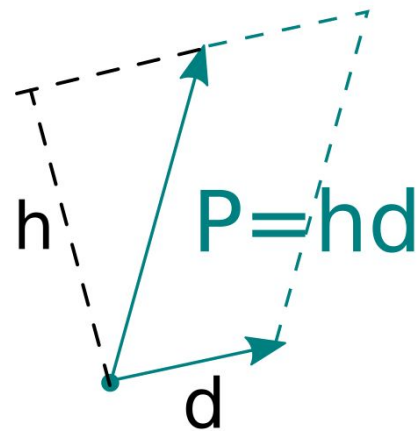
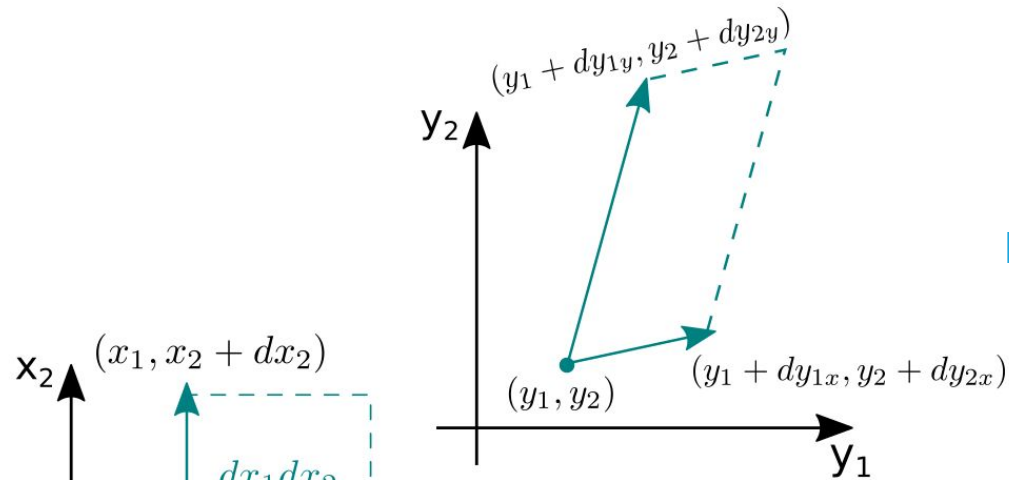
- Similarly for y component:

$$(x_1, x_2 + dx_2) \rightarrow \left(y_1 + \frac{\partial f_1}{\partial x_2} dx_2, y_2 + \frac{\partial f_2}{\partial x_2} dx_2 \right)$$



Normalizing flows - determinant - 2D case

- We want to compute the volume/field of the created parallelogram



We can compute the field P, using basic vector calculus:

$$d = \sqrt{a^2 + b^2} \quad h = \left[\frac{(-b, a)}{\sqrt{a^2 + b^2}} \right] \cdot (c, d) = \frac{ad - bc}{\sqrt{a^2 + b^2}}$$

$$P = hd = ad - bc = \left(\frac{\partial f_1}{\partial x_1} \frac{\partial f_2}{\partial x_2} dx_2 dx_1 - \frac{\partial f_2}{\partial x_1} \frac{\partial f_1}{\partial x_2} dx_2 dx_1 \right)$$

$$= \det \begin{pmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_2}{\partial x_1} \\ \frac{\partial f_1}{\partial x_2} & \frac{\partial f_2}{\partial x_2} \end{pmatrix} dx_2 dx_1 = \det \mathbf{J}^T dx_2 dx_1 = \det \mathbf{J} dx_2 dx_1$$

$\mathbf{J} = \text{Jacobian}$

Normalizing flows - determinant - 2D case

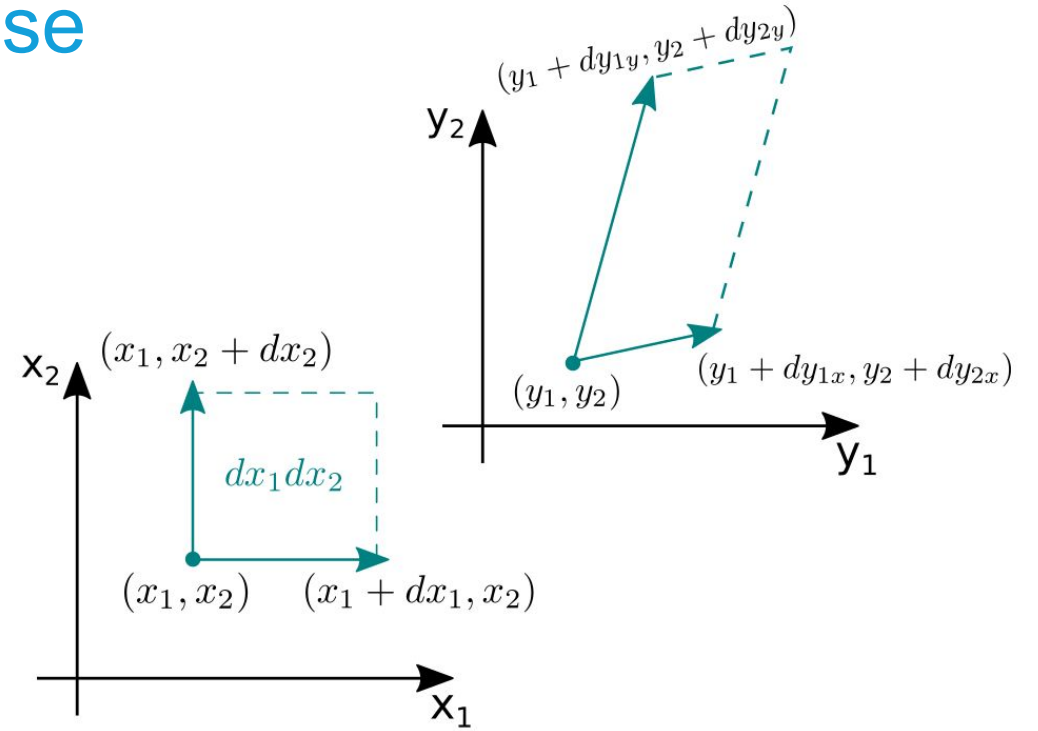
- Finally want to compute:

$$p(\mathbf{y}) = p(\mathbf{x}) \left| \frac{dV_{\mathbf{y}}}{dV_{\mathbf{x}}} \right|^{-1}$$

- where:

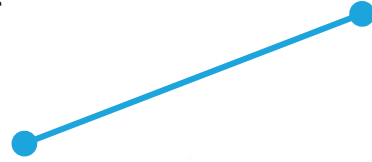
$$dV_{\mathbf{x}} = dx_1 dx_2 \quad dV_{\mathbf{y}} = \det \mathbf{J} dx_2 dx_1$$

- which gives: $p(\mathbf{y}) = p(\mathbf{x}) |\det \mathbf{J}|^{-1}$

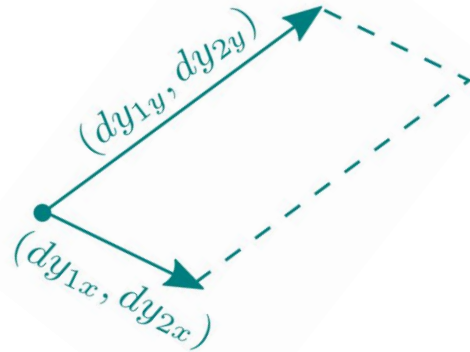


Normalizing flows - determinants summary

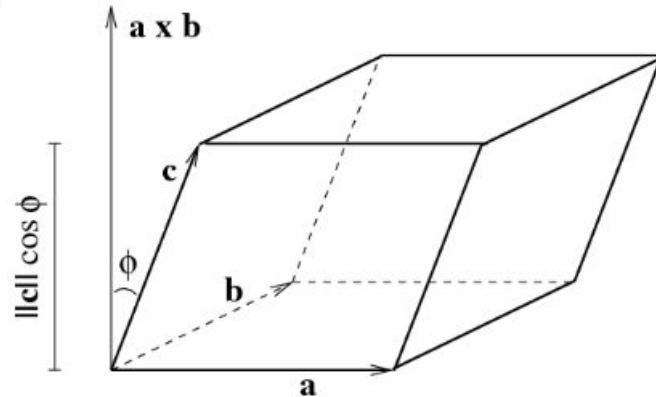
- In 1D we have a Segment



- In 2D Parallelogram



- In 3D Parallelepiped



- In N-D we have n-parallelotope

$$\frac{\partial f(\mathbf{z})}{\partial \mathbf{z}} = \begin{pmatrix} \frac{\partial f_1}{\partial z_1} & \dots & \frac{\partial f_1}{\partial z_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_n}{\partial z_1} & \dots & \frac{\partial f_n}{\partial z_n} \end{pmatrix}$$

$$q'(\mathbf{z}') = q(\mathbf{z}) \left| \det \frac{\partial f(\mathbf{z}^{(n)})}{\partial \mathbf{z}^{(n)}} \right|^{-1}$$

In our case determinant can be understood as a volume of n-parallelotope spanned by the rows (or columns) of the Jacobian matrix.

Normalizing flows - technical details

- So far we have discussed the theoretical aspects of NFs, and we got the formula for probability density flow

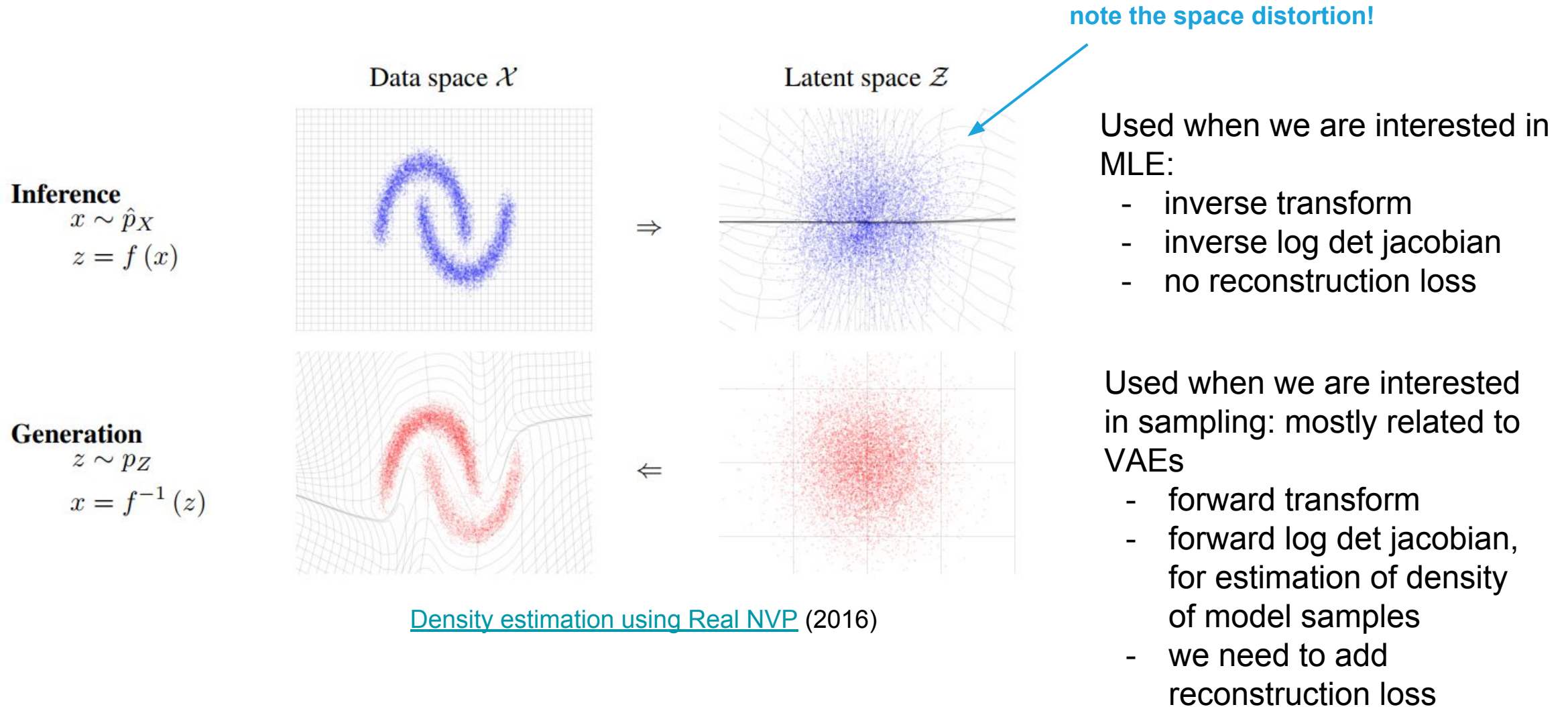
$$\begin{aligned} \mathbf{z}^{(0)} &\sim p(\mathbf{z}) \\ \mathbf{z}^{(1)} &= f^{(1)}(\mathbf{z}^{(0)}) \\ \mathbf{z}^{(2)} &= f^{(2)}(\mathbf{z}^{(1)}) \\ &\dots \\ \mathbf{z}^{(n)} &= f^{(n)}(\mathbf{z}^{(n-1)}) \end{aligned} \quad \longrightarrow \quad q^{(n)}(\mathbf{z}^{(n)}) = q(\mathbf{z}^{(0)}) \prod_{i=1}^n \left| \det \frac{\partial \mathbf{z}^{(i)}}{\partial \mathbf{z}^{(i-1)}} \right|^{-1}$$

- In most cases when people implement NFs they take into account following rules:
 - use **log densities**, since this is more **stable numerically**
 - use f such that we can **compute determinant easily**: diagonal or triangular matrix for which determinant can be easily computed analytically, or other not investigated options ???
 - use f such that computation of **$\mathbf{f}(\mathbf{x})$ is easy**: efficient sampling
 - use f such that computation of **$\mathbf{f}^{-1}(\mathbf{y})$ is easy**: efficient estimation of log-likelihood
 - use f such that we **can compute derivative of it**: can be used with gradient descent
 - use f such that computation is **stable numerically**
 - use f such that it has **high expressive power** usually some non linear function

Note: Various methods have been developed, but usually only part of these conditions are satisfied.

Normalizing flows - inference and generation

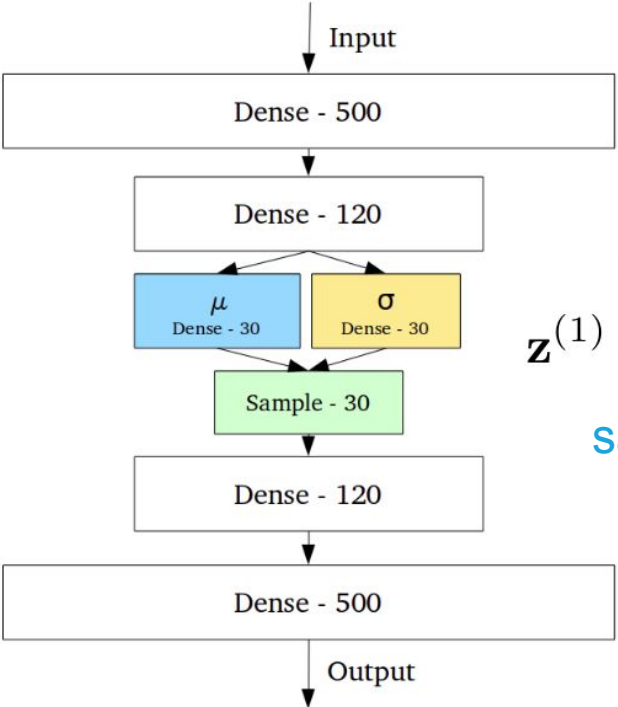
- Let us recall once again difference between inference and generation



Normalizing flows - case study: Variational Inference in VAEs

- [Variational Inference with Normalizing Flows](#) work improves standard VAEs by introducing sampling from approximated posterior generated by NF

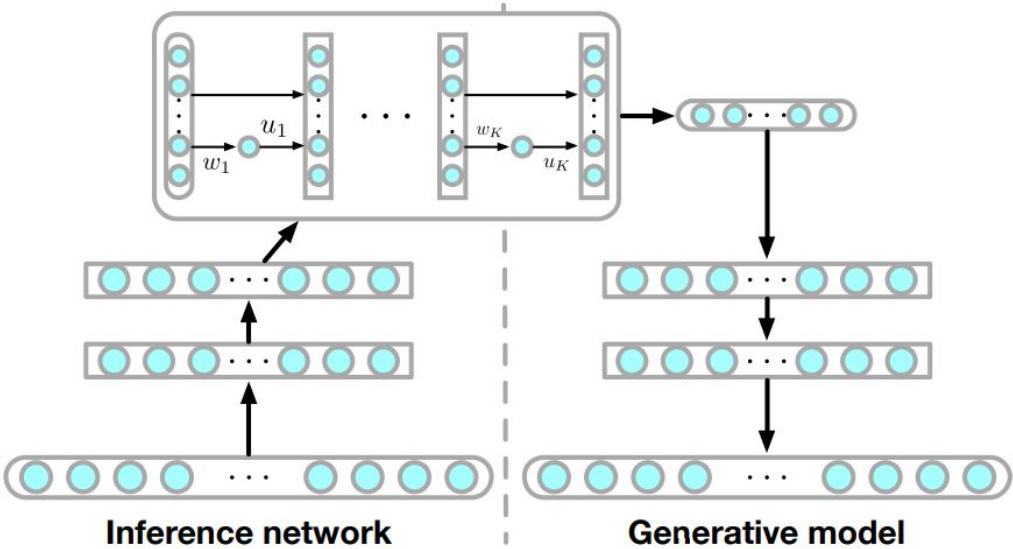
Standard VAE [source](#)



$$\mathbf{z}^{(0)} \sim \mathcal{N}(0, 1)$$
$$\mathbf{z}^{(1)} = \mu(\mathbf{x}) + \mathbf{z}^{(0)} \sigma(\mathbf{x})$$

sampling in VAE

VAE with NF



Standard **VAE** can be understood as special case of Normalizing Flows

$$\mathbf{z}^{(n)} = f^{(n)} \circ f^{(n-1)} \circ \dots \circ f^{(1)} \left(\mathbf{z}^{(0)}; \mathbf{x} \right)$$

sampling in VAE with NF

In VAEs we are interested in forward flow, so e.g. we can resign from easily invertible f

Normalizing flows - case study: Variational Inference in VAEs

- [Variational Inference with Normalizing Flows](#) work improves standard VAEs by introducing sampling from approximated posterior generated by NF

In VAEs we are interested in forward flow, so e.g. we can resign from easily invertible f in favor of more expressive nonlinearity:

Planar flow:

$$f(\mathbf{z}) = \mathbf{z} + \mathbf{u}h(\mathbf{w}^\top \mathbf{z} + b), \quad (10)$$

where $\lambda = \{\mathbf{w} \in \mathbb{R}^D, \mathbf{u} \in \mathbb{R}^D, b \in \mathbb{R}\}$ are free parameters and $h(\cdot)$ is a smooth element-wise non-linearity, $h(x)=\tanh(x)$

Radial flow:

$$f(\mathbf{z}) = \mathbf{z} + \beta h(\alpha, r)(\mathbf{z} - \mathbf{z}_0),$$

where $r = |\mathbf{z} - \mathbf{z}_0|$, $h(\alpha, r) = 1/(\alpha + r)$, and the parameters of the map are $\lambda = \{\mathbf{z}_0 \in \mathbb{R}^D, \alpha \in \mathbb{R}^+, \beta \in \mathbb{R}\}$.

Note: This function cannot be easily inverted, but we just need have it as bijective function (invertibility condition). You can find more information in the paper.

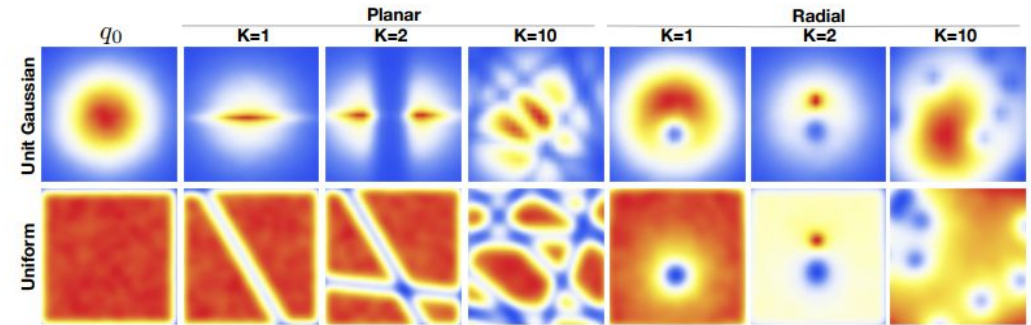
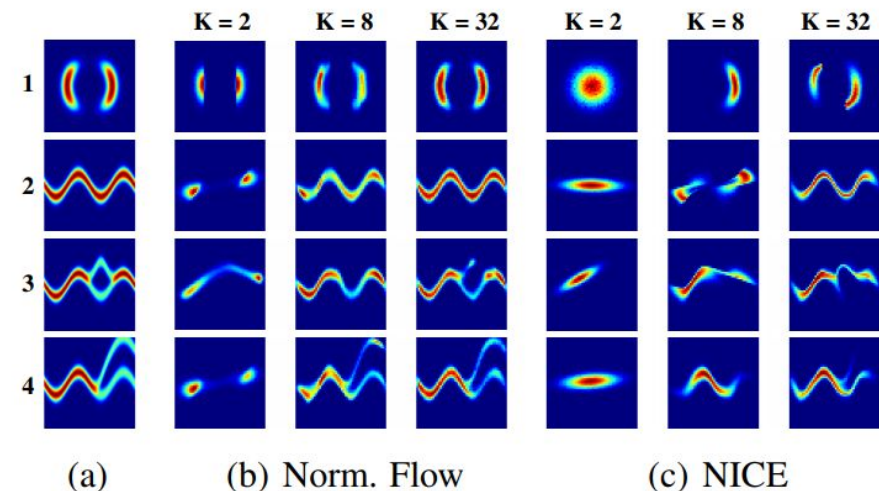


Figure 1. Effect of normalizing flow on two distributions.

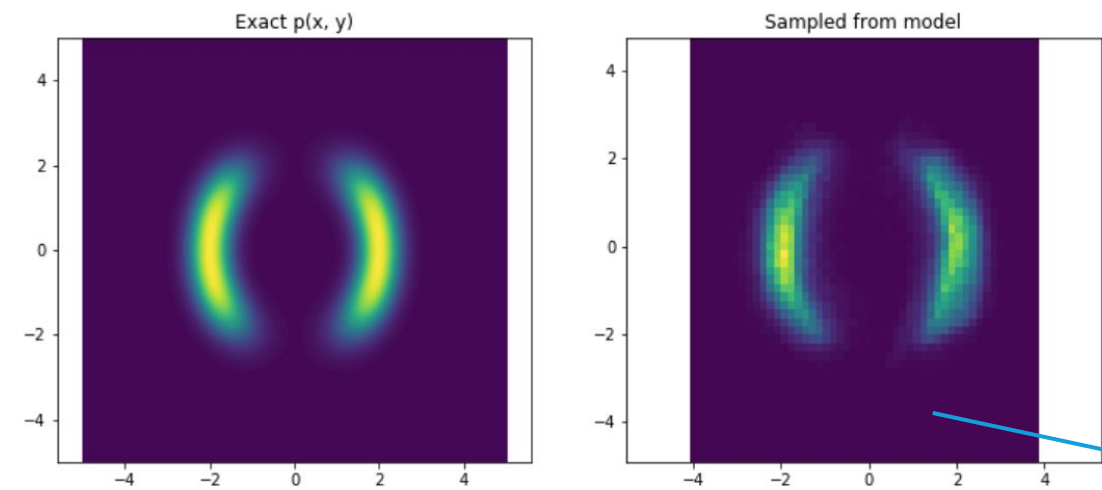
A demonstration of the expressive power of NF with flow length K



Normalizing flows - case study: Variational Inference in VAEs

- [Variational Inference with Normalizing Flows](#) work improves standard VAEs by introducing sampling from approximated posterior generated by NF

jupyter 3_Planar_Flows_experiments



My implementation, flow with K=16

Examples in notebook: 3

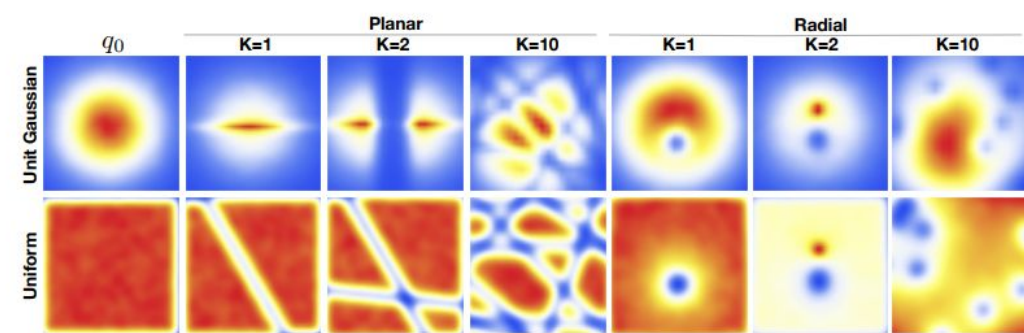
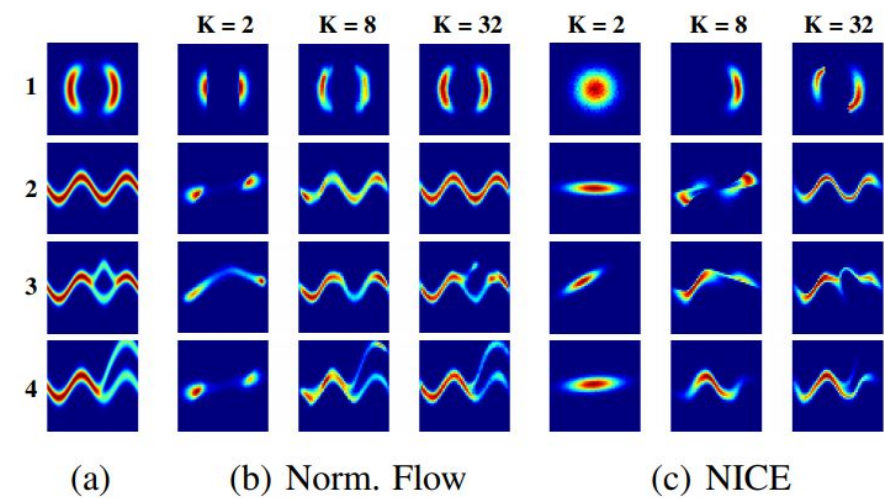


Figure 1. Effect of normalizing flow on two distributions.

A demonstration of the expressive power of NF with flow length K



A short break

Examples: Normalizing Flows in Tensorflow

Normalizing flows - Bijections

Link to publication: <https://arxiv.org/pdf/1711.10604.pdf>

- Normalizing flows in TF are realized with **Bijections** and **Distribution** classes.
- Documentation is quite good.
- Bijections are used to transform some base/simple distribution e.g. univariate Gaussian
- They implement following methods:
 - **forward** - sampling
 - **forward_log_det_jacobian** - density of own sample
 - **inverse** - density estimation
 - **inverse_log_det_jacobian** - density estimation
- Basic properties:

```
x = ... # A tensor.  
# Evaluate forward transformation.  
fwd_x = my_bijection.forward(x)  
x == my_bijection.inverse(fwd_x)  
x != my_bijection.forward(fwd_x) # Not equal because x != g(g(x)).
```

- Computing a log-likelihood:

```
def transformed_log_prob(bijection, log_prob, x):  
    return (bijection.inverse_log_det_jacobian(x, event_ndims=0) +  
            log_prob(bijection.inverse(x)))
```

Forward sampling:

$$\log q^{(n)}(\mathbf{z}^{(n)}) = \log q(\mathbf{z}^{(0)}) - \sum_{i=1}^n \log \left| \det \frac{\partial \mathbf{z}^{(i)}}{\partial \mathbf{z}^{(i-1)}} \right|$$

- Transforming a random outcome:

```
def transformed_sample(bijection, x):  
    return bijection.forward(x)
```

Normalizing flows - Bijectors

- Sample-Batch-Event shape tensors name semantics in Bijectors

n Monte Carlo draws	b examples per batch	s latent dimensions
sample_shape (indep, identically distributed)	batch_shape (indep, <i>not</i> identical)	event_shape (can be dependent)

Figure 4. Shape semantics. Refers to variational distribution in Figure 1.

- Sampling from distributions

```
base_dist = tfd.MultivariateNormalDiag(loc=tf.zeros([2, 4], tf.float32))
base_dist.sample()           # [2, 4]
base_dist.sample(3)          # [3, 2, 4]
base_dist.sample([3, 5])     # [3, 5, 2, 4]
```

- Sampling from transformed distribution

```
dist = tfd.TransformedDistribution(
    distribution=base_dist,
    bijector=tfb.Sigmoid(),
)

dist.sample()           # [2, 4]
dist.sample(3)          # [3, 2, 4]
dist.sample([3, 5])     # [3, 5, 2, 4]
```

```
base_dist = tfd.Normal(loc=0.0, scale=1.0) # must be scalar
dist = tfd.TransformedDistribution(
    batch_shape=[2],
    event_shape=[4],
    distribution=base_dist,
    bijector=tfb.Sigmoid(),
)

dist.sample()           # [2, 4]
dist.sample(3)          # [3, 2, 4]
dist.sample([3, 5])     # [3, 5, 2, 4]
```

Normalizing flows - simple Exp Bijector

- Taken from Tensorflow [documentation](#)

For custom Bijectors we must implement `_private methods`

```
class Exp(Bijector):
```

```
    def __init__(self, validate_args=False, name="exp"):
        super(Exp, self).__init__(
            validate_args=validate_args,
            forward_min_event_ndims=0,
            name=name)
```

```
    def _forward(self, x):
        return math_ops.exp(x)
```

```
    def _inverse(self, y):
        return math_ops.log(y)
```

```
    def _inverse_log_det_jacobian(self, y):
        return -self._forward_log_det_jacobian(self._inverse(y))
```

```
    def _forward_log_det_jacobian(self, x):
        # Notice that we needn't do any reducing, even when `event_ndims > 0`.
        # The base Bijector class will handle reducing for us; it knows how
        # to do so because we called `super` `__init__` with
        # `forward_min_event_ndims = 0`.
        return x
```

```
...
```

$$\mathbf{y} = \exp(\mathbf{x})$$

$$\mathbf{x} = \log(\mathbf{y})$$

$$\mathbf{J}_{ij} = \exp(x_i) \delta_{i,j}$$

$$\det(\mathbf{J}_{ij}) = \prod_i \exp(x_i)$$

$$\log \det(\mathbf{J}_{ij}) = \sum_i x_i$$

Inverse function theorem

[A general mathematical property](#)

$$\text{inv det}(\mathbf{J}_{ij}) = \prod_i \frac{1}{y_i}$$

$$\text{inv log det}(\mathbf{J}_{ij}) = - \sum_i \log y_i = - \sum_i x_i$$

Normalizing flows - Chain Bijector

- Bijections can be chained with **Chain Bijector**.
- This allows to compose complicated distributions
- We can **sample** from them or even compute **log_prob**, **prob** values

Example Use:

```
chain = Chain([Exp(), Softplus()], name="one_plus_exp")
```

Results in:

- Forward:

```
exp = Exp()
softplus = Softplus()
Chain([exp, softplus]).forward(x)
= exp.forward(softplus.forward(x))
= tf.exp(tf.log(1. + tf.exp(x)))
= 1. + tf.exp(x)
```

```
dist = tfd.TransformedDistribution(
    distribution=base_dist,
    bijector=tfb.Chain(list(reversed(bijectors))),
)
```

Note: Remember to reverse your bijectors list, since the bijectors are applied from the end, as is done in mathematical equation:

$$\mathbf{z}^{(n)} = f^{(n)} \circ f^{(n-1)} \circ \dots \circ f^{(1)} \left(\mathbf{z}^{(0)}; \mathbf{x} \right)$$

Normalizing flows - example: estimating density

```
1 base_dist = tfd.MultivariateNormalDiag(loc=tf.zeros([2], tf.float32))
2 bijectors = [
3     tfb.AffineScalar(shift=tf.constant([0.2, 2.0]), scale=tf.constant([0.3, 1.2])),
4     tfb.Sigmoid()
5 ]
6
7 dist = tfd.TransformedDistribution(
8     distribution=base_dist,
9     bijector=tfb.Chain(list(reversed(bijectors))),
10 )
```

Affine scalar: μ , σ are
acts elements wise

$$x = \mu + \varepsilon\sigma$$

- We want to estimate the density at some point \mathbf{z}
- This can be done easily with Bijectors by calling **log_prob** or **prob**
- **log_prob**

```
1 dist.log_prob(z).eval()
array([-0.83020973, -2.2573261 ], dtype=float32)
```

- **prob**

```
1 dist.prob(z).eval()
array([0.43595785, 0.10462989], dtype=float32)
```


Some people prefer manual computation:

```
1 # some initial z
2 z = tf.constant([0.6, 0.5])
3
4 # inverse first bijector + accumulate log_det_jacobian
5 z_1 = bijectors[1].inverse(z)
6 log_z1_z = bijectors[1].inverse_log_det_jacobian(z)
7
8 # inverse second bijector + accumulate log_det_jacobian
9 z_0 = bijectors[0].inverse(z_1)
10 log_z0_z1 = bijectors[0].inverse_log_det_jacobian(z_1)
11
12 # final density
13 log_prob = base_dist.log_prob(z_0) + log_z1_z + log_z0_z1
```

Normalizing flows - tensorflow probability package


More examples at: https://github.com/tensorflow/probability/tree/master/tensorflow_probability/examples

Branch: master ▾ **probability** / tensorflow_probability / **examples** / Create new file Upload


 **Copybara-Service** Merge pull request #111 from jmzeng:bayesianCNNs ...

Latest


..

 [jupyter_notebooks](#)


Add colab buttons and !pip install

 [BUILD](#)


Temporarily disable bayesian neural network test to unbreak continuou...

 [bayesian_neural_network.py](#)


Merge pull request #111 from jmzeng:bayesianCNNs

 [latent_dirichlet_allocation.py](#)


Update README.md with links to all notebooks & example scripts.

 [logistic_regression.py](#)

Import distributions from tensorflow_probability, not tensorflow.

 [vae.py](#)

Merge pull request #123 from m-colombo:patch-1

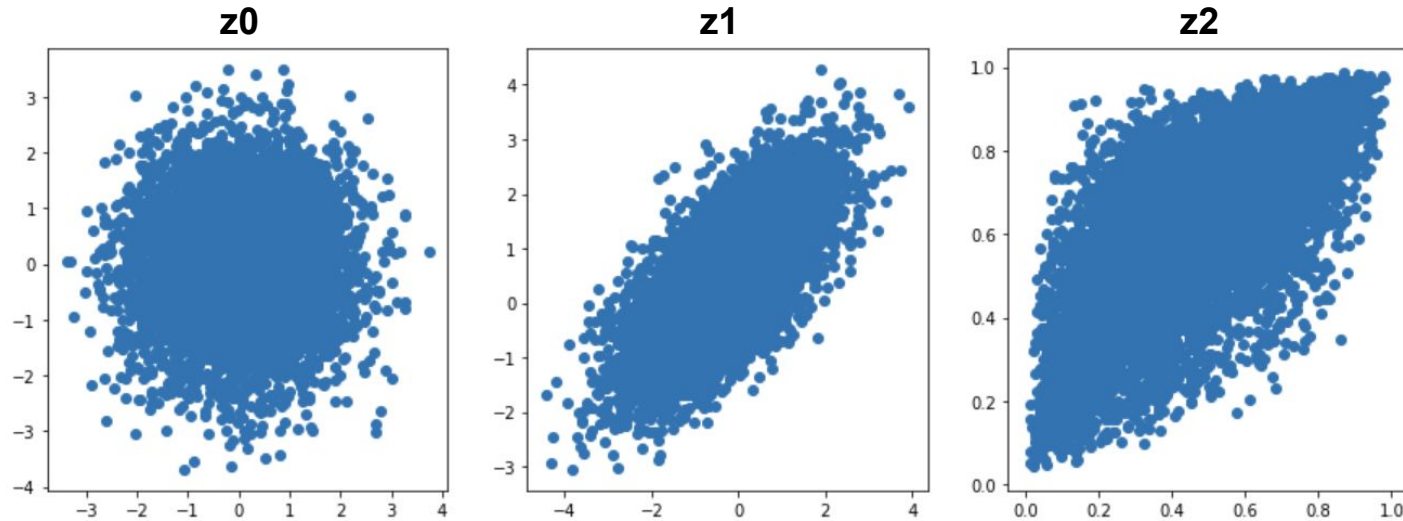
 [vq_vae.py](#)

Use make_{decoder,encoder} practice following vae.py. Clean up docs.

Density estimation: maximum likelihood method

- With NFs we can do exact MLE, since likelihood is trackable
- Consider following example:

```
z0 = np.random.randn(*[32 * 200, 2])  
z1 = z0 @ np.array([[0.5, 1.0], [1, .3]]) + np.array([-0.2, .5])  
z2 = 1 / (1 + np.exp(- z1))
```



In MLE we want to maximize ($X=z2$): $\log P_{\text{model}}(\mathbf{X}; \theta)$

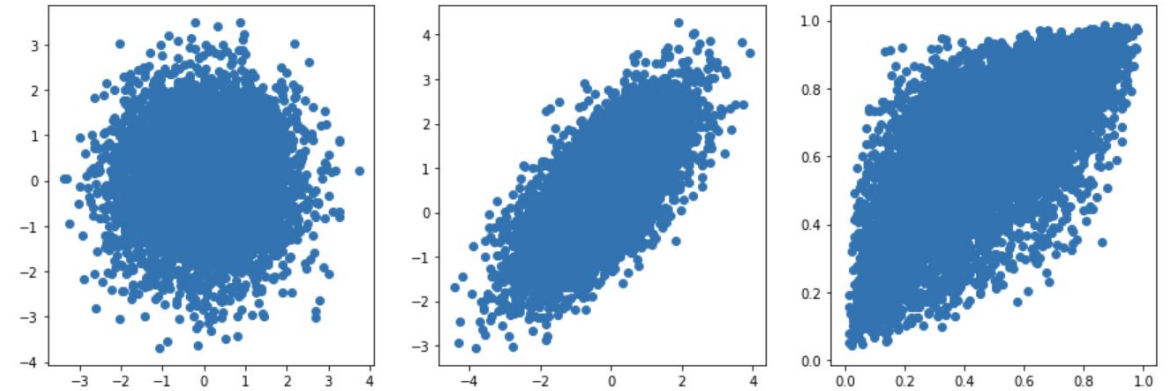
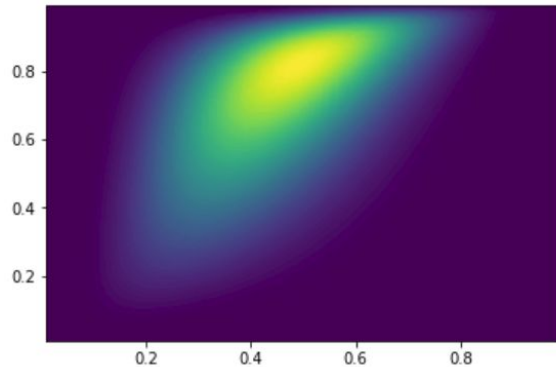
Density estimation: maximum likelihood method - demo

- Let's build probabilistic P_{model} for which we can sample and estimate $\log P_{\text{model}}$
 - In real world problems we have access to z_2 only
- Model definition:

```
base_dist = tfd.MultivariateNormalDiag(loc=tf.zeros([2], tf.float32))
bijectors = [
    tfb.Affine(
        scale_tril=tf.get_variable("tril", shape=[2, 2]),
        shift=tf.get_variable("shift", shape=[2])
    ),
    Sigmoid()
]

dist = tfd.TransformedDistribution(
    distribution=base_dist,
    bijector=tfb.Chain(list(reversed(bijectors))),
)
```

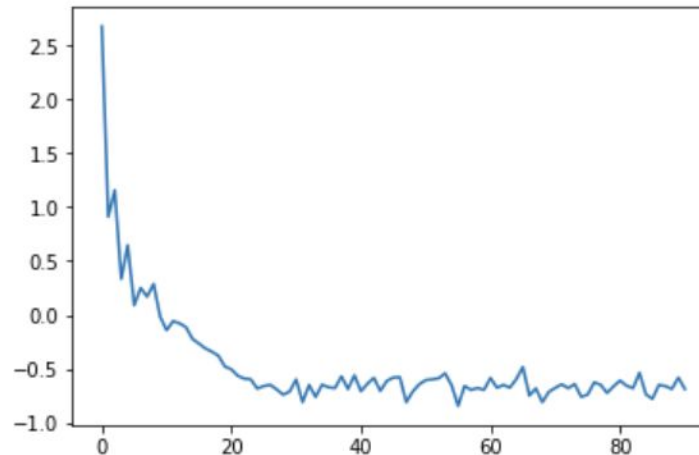
- Initial samples from the model:



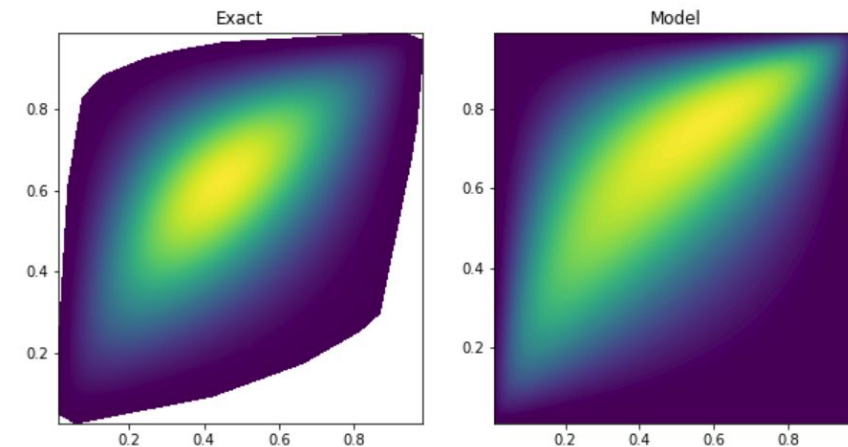
- Definition of the cost function: $L = -\log p(z_2)$

```
1 loss = -tf.reduce_mean(dist.log_prob(z2_samples))
2 train_op = tf.train.AdamOptimizer(1e-3).minimize(loss)
```

- Training:



- Learned samples:

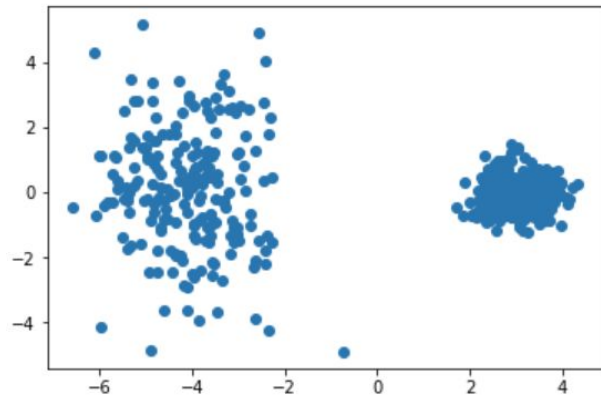


Density estimation: MLE and mixture of gaussians (not NFs)

- Demo in the notebook

2. Model definition

1. Toy dataset:

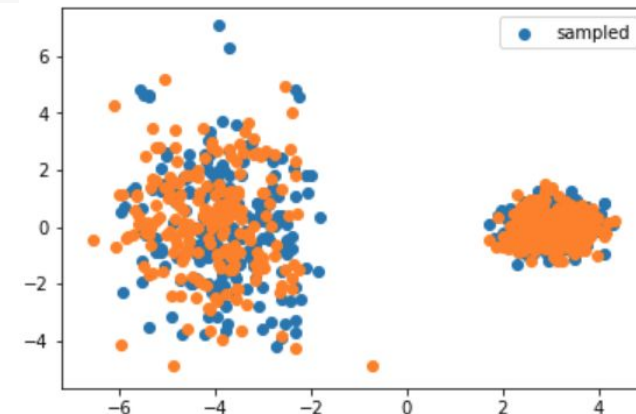


```
1 probs = tf.get_variable('mixture_probs', [2], dtype=tf.float32)
2 # keep probs normalized
3 probs = tf.nn.softmax(probs)
4
5 scale_fn = lambda name: tf.nn.softplus(tf.get_variable(name, initializer=np.ones([2], dtype='float32')))
6
7 model_dist = tfd.Mixture(
8     cat = tfd.Categorical(probs=probs),
9     components = [
10         tfd.MultivariateNormalDiag(
11             loc=tf.get_variable('loc1', [2]),
12             scale_diag=scale_fn('scale1')
13         ),
14         tfd.MultivariateNormalDiag(
15             loc=tf.get_variable('loc2', [2]),
16             scale_diag=scale_fn('scale2')
17         )
18     ]
19 )
```

3. loss function: $L = -\frac{1}{N_B} \sum_k \log(p(x))$

```
1 loss = - tf.reduce_mean(model_dist.log_prob(x_samples))
2 train_op = tf.train.AdamOptimizer(1e-3).minimize(loss)
```

4. Samples from trained model



Density estimation: summary

For simple models we can just call to define cost function

```
loss = - tf.reduce_mean(model_dist.log_prob(x_samples))
```

x_samples - our data or minibatch e.g. tensor of shape [32, 2]

loss - the current value maximum likelihood we want to minimize

model_dist - some probabilistic model

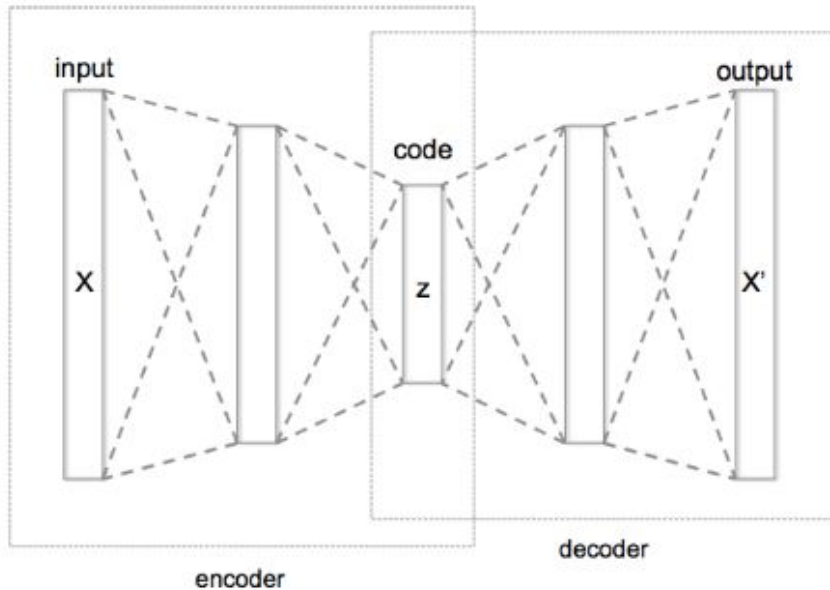
For hard problems this will usually require some hacks and practice...

Part 2. Practical implementations of NFs with Deep Neural Networks

NFs with Neural Networks:
MADE: Masked Autoencoder for Distribution
Estimation (2015)

Autoregressive models - MADE

- MADE - Masked Autoencoder for Distribution Estimation
- MADE is a building block of many NFs architectures, but itself is not related with NFs
- People use this concept because it allows for creating functions with triangular Jacobian matrix
- We start with high level description of the problem considered in MADE paper.
- The authors consider autoencoder with binary cross entropy loss



$$\begin{aligned}\mathbf{h}(\mathbf{x}) &= \mathbf{g}(\mathbf{b} + \mathbf{W}\mathbf{x}) \\ \hat{\mathbf{x}} &= \text{sigm}(\mathbf{c} + \mathbf{V}\mathbf{h}(\mathbf{x})),\end{aligned}$$

To train the autoencoder, we must first specify a training loss function. For binary observations, a natural choice is the cross-entropy loss:

$$\ell(\mathbf{x}) = \sum_{d=1}^D -x_d \log \hat{x}_d - (1 - x_d) \log(1 - \hat{x}_d). \quad (3)$$

This loss result from log of Bernoulli distribution assumption:

$$q(\mathbf{x}) = \prod_d \hat{x}_d^{x_d} (1 - \hat{x}_d)^{1 - x_d}$$

This final loss can be zero for some trivial case with $q(\mathbf{x}) = 1$, hence $q(\mathbf{x})$ is not a proper since:

$$\sum_{\mathbf{x}} q(\mathbf{x}) \neq 1$$


Autoregressive model - MADE

- In short: vanilla autoencoder with binary cross entropy loss, does not have probabilistic interpretation.
- The authors want to make it a probabilistic model with trackable and normalized $p(\mathbf{x})$.
- They propose following solution for that:

First, we can use the fact that, for any distribution, the probability product rule implies that we can always decompose it into the product of its nested conditionals

$$p(\mathbf{x}) = \prod_{d=1}^D p(x_d \mid \mathbf{x}_{<d}), \quad (4)$$

Probabilistic chain rule
e.g. $P(x, y) = P(y \mid x) * P(x)$



where $\mathbf{x}_{<d} = [x_1, \dots, x_{d-1}]^\top$.

$$\begin{aligned} \ell(\mathbf{x}) = \sum_{d=1}^D & -x_d \log p(x_d = 1 \mid \mathbf{x}_{<d}) \\ & - (1 - x_d) \log p(x_d = 0 \mid \mathbf{x}_{<d}) \end{aligned}$$

Autoregressive model - MADE

- Formally, autoencoder forms a proper distribution if each output unit d depends on units $d' < d$.
- They call this **autoregressive property**, since the negative log-likelihood is equivalent to sequentially predicting (regressing) each dimension of input \mathbf{x}
- The last sentence means that the **Jacobian** matrix of the autoencoder is **lower triangular matrix** with zeros on the diagonal:

output vector of the autoencoder $\rightarrow \frac{\partial \hat{\mathbf{x}}}{\partial \mathbf{x}}$

input vector of the autoencoder $\rightarrow \mathbf{x}$

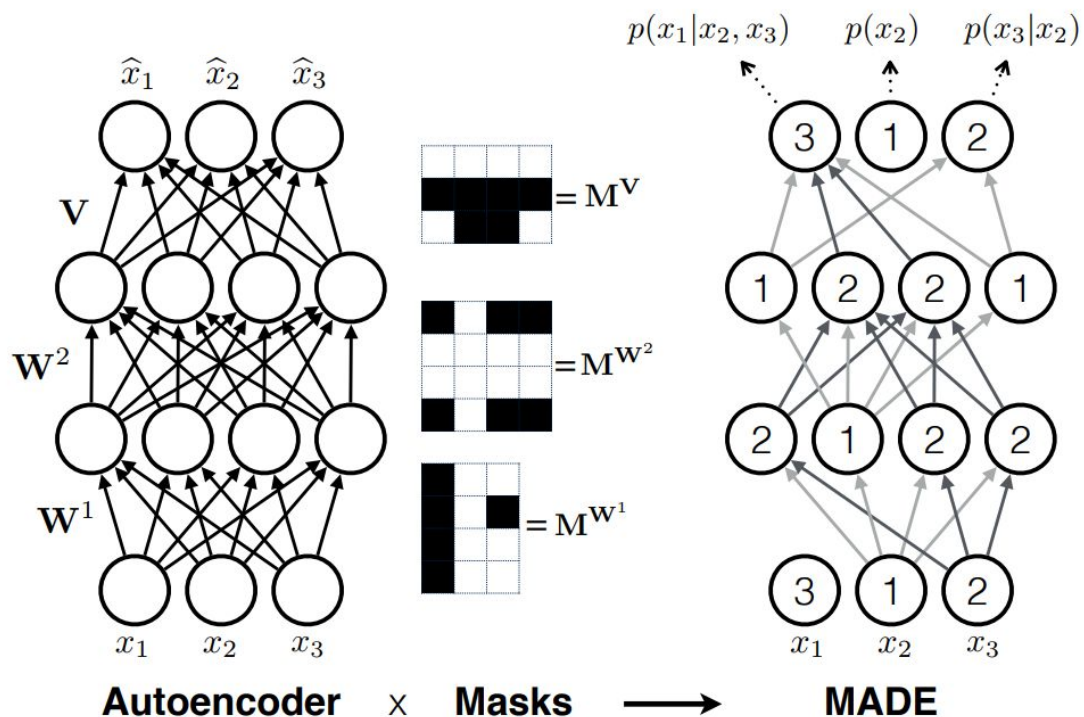
$$\frac{\partial \hat{\mathbf{x}}}{\partial \mathbf{x}} = \begin{pmatrix} 0 & \dots & & \\ \frac{\partial \hat{x}_1}{\partial x_0} & 0 & & \\ \vdots & & \ddots & \vdots \\ \frac{\partial \hat{x}_n}{\partial x_0} & \dots & \frac{\partial \hat{x}_n}{\partial x_{n-1}} & 0 \end{pmatrix}$$

- This property is achieved by masking weights of the autoencoder matrices

$$\begin{aligned} \mathbf{h}(\mathbf{x}) &= \mathbf{g}(\mathbf{b} + (\mathbf{W} \odot \mathbf{M}^{\mathbf{W}})\mathbf{x}) \\ \hat{\mathbf{x}} &= \text{sigm}(\mathbf{c} + (\mathbf{V} \odot \mathbf{M}^{\mathbf{V}})\mathbf{h}(\mathbf{x})) \end{aligned}$$

Autoregressive model - MADE

- MADE:



Copied from [A. Karpathy](#) MADE implementation

```
def create_masks(
    input_size,
    hidden_sizes=[500],
    natural_ordering=True,
    seed=4198721
):
    rng = np.random.RandomState(seed)
    L = len(hidden_sizes)
    m = {}
    # sample the order of the inputs and the connectivity of all neurons
    m[-1] = np.arange(input_size) if natural_ordering else rng.permutation(input_size)
    for l in range(L):
        m[l] = rng.randint(m[l-1].min(), nin-1, size=hidden_sizes[l])

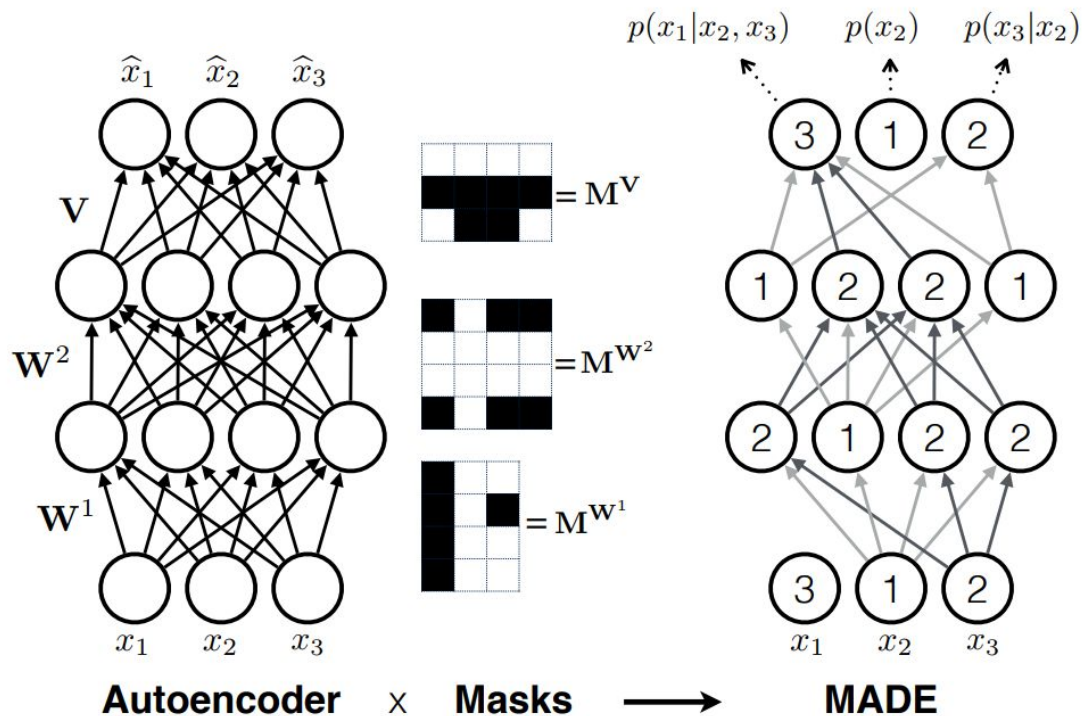
    # construct the mask matrices
    masks = [(m[l-1][:,None] <= m[l][None,:])*1 for l in range(L)]
    masks.append((m[L-1][:,None] < m[-1][None,:])*1)
    return masks
```

13 lines for simple MADE implementation:

```
1 hidden_size = 500
2 input_size = 784
3
4 # define masks
5 masks = create_masks(input_size, [hidden_size])
6 # define simple model
7 x_ph = tf.placeholder(tf.float32, [batch_size, input_size])
8 hidden = masked_dense(x_ph, hidden_size, mask=masks[0])
9 x_hat = masked_dense(hidden, input_size, mask=masks[1], activation=None)
10 # loss
11 loss = tf.nn.sigmoid_cross_entropy_with_logits(
12     labels=x_ph,
13     logits=x_hat) / batch_size
```

MADE - final remarks

- MADE:

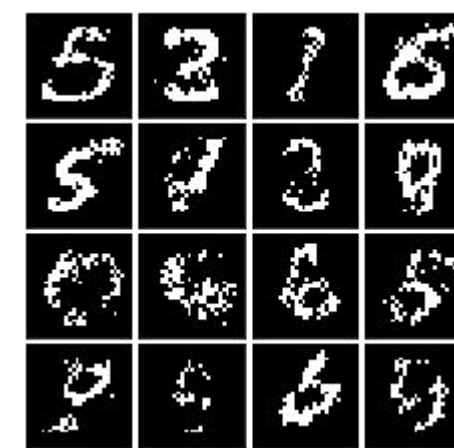


Some final remarks:

- MADE allows to create a NN which Jacobian matrix will be lower triangular with zeros on diagonal
- MADE makes autoencoder probabilistic model from which one can sample:



From paper



My samples 2 obtained from hidden layers model

Bad things:

- Single mask works poorly**, so they need to run model on ensemble of different mask to reach SOTA, i.e. **slower training, slower sampling** etc...
- Masks do not depend on data**, instead they depend on some heuristics which may make MADEs just worse

MADE - final final remarks

- Watch out on different MADE implementations! e.g. Tensorflow implementation leads to deterministic matrices and **num_blocks** is not defined in the original paper and it is not documented in TF implementation.

```
1 from tensorflow.contrib.distributions.python.ops.bijectors.masked_autoregressive import _gen_mask
2 |
3 mask = _gen_mask(num_blocks=10, n_in=15, n_out=10, mask_type='exclusive')
4 mask
: array([[0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
        [1., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
        [1., 1., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
        [1., 1., 1., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
        [1., 1., 1., 1., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
        [1., 1., 1., 1., 1., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
        [1., 1., 1., 1., 1., 1., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
        [1., 1., 1., 1., 1., 1., 1., 0., 0., 0., 0., 0., 0., 0., 0.],
        [1., 1., 1., 1., 1., 1., 1., 1., 0., 0., 0., 0., 0., 0., 0.],
        [1., 1., 1., 1., 1., 1., 1., 1., 1., 0., 0., 0., 0., 0., 0.],
        [1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 0., 0., 0., 0., 0.]],
        dtype=float32)
```

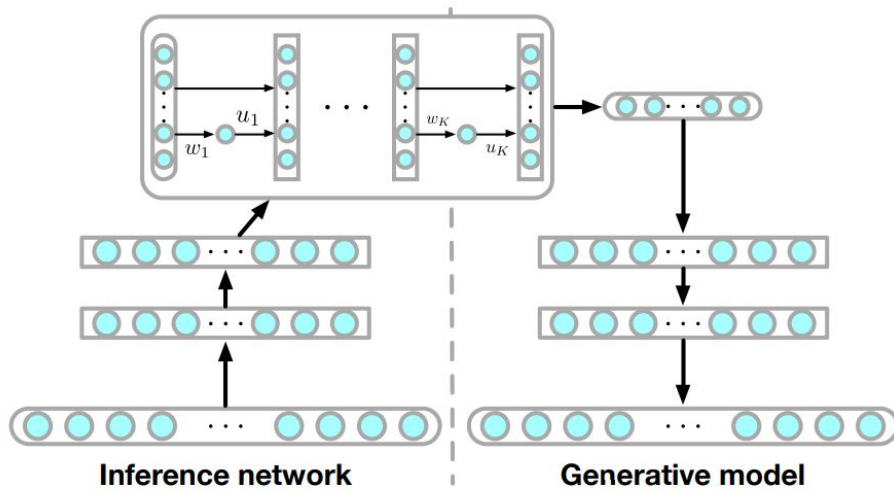
- This function is used in `masked_dense` function which is used by few other functions which are used to build some NF networks.
- The [original implementation](#) of the mask generation has ~100 lines and it is hard to read and it is written in Theano.
- [Linear mask implementation](#) from IAF paper is also different.
- Autoregressive masks can be implemented for convolutional layers ([check this](#))

Improved Variational Inference with Inverse Autoregressive Flow (**OpenAI/2016**)

Normalizing flows + VAE + MADE

- We discuss the content of this paper: [Improved Variational Inference with Inverse Autoregressive Flow](#)
- We already discussed the application of NFs to VAEs

VAE with NF



- The authors of IAF argue that **planar and radial flows** used by (Rezende and Mohamed 2015) have **limited expressive power**
- They propose to use **MADE blocks**.
- Remember: In **VAEs** we are interested in **forward flow**, so e.g. we can resign from easily invertible f

$$\mathbf{z}^{(0)} \sim \mathcal{N}(0, 1)$$

$$\mathbf{z}^{(1)} = \mu(\mathbf{x}) + \mathbf{z}^{(0)} \sigma(\mathbf{x})$$

$$\mathbf{z}^{(n)} = f^{(n)} \circ f^{(n-1)} \circ \dots \circ f^{(1)} \left(\mathbf{z}^{(0)}; \mathbf{x} \right)$$

IAF - Inverse Autoregressive Flow

- The algorithm for IAF is following:

$[\mu, \sigma, \mathbf{h}] \leftarrow \text{EncoderNN}(\mathbf{x}; \theta)$

$\epsilon \sim \mathcal{N}(0, I)$

$\mathbf{z} \leftarrow \sigma \odot \epsilon + \mu$

$l \leftarrow -\text{sum}(\log \sigma + \frac{1}{2} \epsilon^2 + \frac{1}{2} \log(2\pi))$

for $t \leftarrow 1$ **to** T **do**

$[\mathbf{m}, \mathbf{s}] \leftarrow \text{AutoregressiveNN}[t](\mathbf{z}, \mathbf{h}; \theta)$

$\sigma \leftarrow \text{sigmoid}(\mathbf{s})$

$\mathbf{z} \leftarrow \sigma \odot \mathbf{z} + (1 - \sigma) \odot \mathbf{m}$

$l \leftarrow l - \text{sum}(\log \sigma)$

end

initialize the chain of transformations by taking the output of the encoder network.

compute new mean and logits with **AutoregressiveNN**

apply NF transformation (stable version)

aggregate log densities

- Final density:
$$\log q(\mathbf{z}_T | \mathbf{x}) = - \sum_{i=1}^D \left(\frac{1}{2} \epsilon_i^2 + \frac{1}{2} \log(2\pi) + \sum_{t=0}^T \log \sigma_{t,i} \right)$$

- VAE loss:
$$\mathbb{E}_{q(\mathbf{z} | \mathbf{x})} [\underbrace{\log p(\mathbf{x}, \mathbf{z})}_{\text{reconstruction term}} - \underbrace{\log q(\mathbf{z} | \mathbf{x})}_{\text{regularization term}}] = \mathcal{L}(\mathbf{x}; \theta)$$

IAF as a Normalizing Flow - Computing Jacobian

Firstly, we show the properties of the jacobian of IAF transformation:

- Consider following IAF transformation (for the brevity we skip the context vector):

$$\mathbf{z}' = \mathbf{z} \cdot \sigma(\mathbf{z}) + \mu(\mathbf{z})$$

- where: $\frac{\partial \sigma(\mathbf{z})_i}{\partial z_j} = 0$ if $i \leq j$ i.e. it is lower triangular matrix with zeros on diagonal. Same for mean.
- Then the Jacobian of this transformation is following:

$$\frac{\partial z'_i}{\partial z_j} = \frac{dz_i}{dz_j} \cdot \sigma(\mathbf{z})_i + z_i \cdot \frac{\partial \sigma(\mathbf{z})_i}{\partial z_j} + \frac{\partial \mu(\mathbf{z})_i}{\partial z_j}$$

$$\frac{\partial z'_i}{\partial z_j} = 0 \text{ if } i < j$$

Jacobian is triangular
with zeros over
diagonal

$$\frac{\partial z'_i}{\partial z_i} = \sigma(\mathbf{z})_i$$

diagonal elements of the
Jacobian

- The log det of the Jacobian is then: $\log \det \mathbf{J} = \sum_i \log \sigma_i$

IAF - Inverse Autoregressive Flow

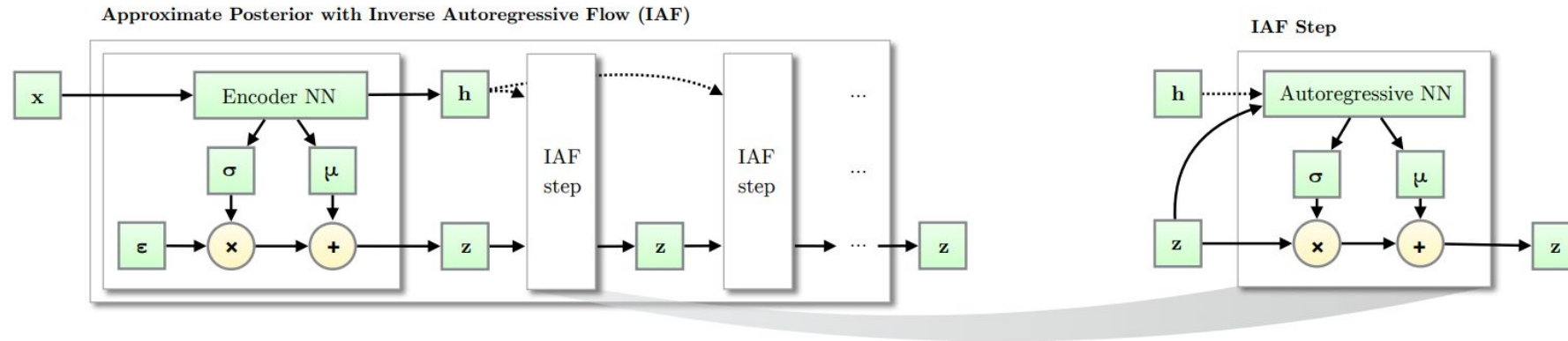


Figure 2: Like other normalizing flows, drawing samples from an approximate posterior with Inverse Autoregressive Flow (IAF) consists of an initial sample z drawn from a simple distribution, such as a Gaussian with diagonal covariance, followed by a chain of nonlinear invertible transformations of z , each with a simple Jacobian determinants.

Some technical remarks and tricks they used:

- They **reverse order** of z after each IAF step - it preserves volume so it does not change jacobian
- They pass **context** vector h between each step, since it helps.
- **AutoregressiveNN** is a few layer Neural Network build on **MADE** idea.
- Sampling is **fast**.
- **Batch normalization** introduced too much noise and replaced with weight normalization.
- **ELU** instead of ReLU and data-dependent initialization
- Custom **modified loss** which blocks oversimplification by the regularization term (see appendix in paper)
- They **did not show image samples**, which would be nice. Numbers can be tricky ...
- They also build a **complicated stochastic ResNet units**, which suggests that the method itself is not that powerful since they need to overcomplicate the architecture to achieve good results.

IAF as a Normalizing Flow

- Note, that NFs are invertible transformations
- IAF involves blocks of NNs which are not invertible

Is IAF invertible?

IAF as a Normalizing Flow - Invertibility of the transformation

- IAF allows us to compute \mathbf{z}' given \mathbf{z} : $\mathbf{z}' = \mathbf{z} \cdot \sigma(\mathbf{z}) + \mu(\mathbf{z})$
- Now, we want to compute \mathbf{z} from \mathbf{z}' : $\mathbf{z} = [\mathbf{z}' - \mu(\mathbf{z})] / \sigma(\mathbf{z})$
- In general sigma and mu are autoregressive NNs which are not invertible.
- Since sigma and mu are autoregressive we can write equation for the first element of \mathbf{z}' vector. Note that sigma and mu cannot depend on z_0 , so I wrote them as some scalars.

$$z'_0 = z_0 \cdot \sigma + \mu \quad \text{from which we can compute } z_0 \text{ as} \quad z_0 = [z'_0 - \mu] / \sigma$$

- Having z_0 we can compute z_1 and so on ...

$$z'_1 = z_1 \cdot \sigma(z_0) + \mu(z_0) \Rightarrow z_1 = [z'_1 - \mu(z_0)] / \sigma(z_0)$$

$$z'_2 = z_2 \cdot \sigma(z_0, z_1) + \mu(z_0, z_1) \Rightarrow z_2 = [z'_2 - \mu(z_0, z_1)] / \sigma(z_0, z_1)$$

$$z'_D = z_D \cdot \sigma(\mathbf{z}_{0:D-1}) + \mu(\mathbf{z}_{0:D-1}) \Rightarrow z_D = [z'_D - \mu(\mathbf{z}_{0:D-1})] / \sigma(\mathbf{z}_{0:D-1})$$

IAF as a Normalizing Flow - Invertibility of the transformation

- Inverse operation is slow, and requires D updates, where D is the dimensionality of the z vector
- Recall, that for sampling (forward pass we need one pass), but here for the evaluation of the density we need D passes.
- In general we can write inverse flow as:

$$z_K = \left[z'_K - \mu(\mathbf{z}_{0:K-1}) \right] / \sigma(\mathbf{z}_{0:K-1})$$

- Which can be conveniently implemented using [simple algorithm](#) (written in Bijectors API):

```
1 def inverse(x):  
2     y = array_ops.zeros_like(x, name="y0")  
3     for _ in range(event_size):  
4         shift, log_scale = self._shift_and_log_scale_fn(y)  
5         y = x * tf.exp(log_scale) + shift  
6     return y
```

$$[\mathbf{m}_t, \mathbf{s}_t] \leftarrow \text{AutoregressiveNN}[t](\mathbf{z}_t, \mathbf{h}; \boldsymbol{\theta})$$

Correction: this is true for MAF
not IAF, for IAF it should be (x-
shift) / sigma

IAF as a Normalizing Flow - my contribution

- This can be further improved by noting that we not necessarily need D iterations:
- This iterative algorithm can reach convergence much faster
- So it is better to use adaptive dynamic loop:

```
1 def inverse(x):
2     y = array_ops.zeros_like(x, name="y0")
3     for _ in range(event_size):
4         shift, log_scale = self._shift_and_log_scale_fn(y)
5         y = x * tf.exp(log_scale) + shift
6     return y
```

- In my tests this implementation was about ten times faster, but it will depend on the problem size and complexity. In some cases convergence may appear after few iterations.

```
1 def adaptive_inverse(x):
2     event_size = x.shape.as_list()[-1]
3     y0 = tf.zeros_like(x, name="y0")
4     # call the template once to ensure creation
5     _ = self._shift_and_log_scale_fn(y0)
6
7     def _loop_body(old_delta, y_prev):
8         shift, log_scale = self._shift_and_log_scale_fn(y_prev)
9         y_new = x * tf.exp(log_scale) + shift
10        delta_y = tf.reduce_max(tf.abs(y_prev - y_new))
11        return delta_y, y_new
12
13    _, y = tf.while_loop(
14        cond=lambda delta_y, _: delta_y > 1e-5,
15        body=_loop_body,
16        loop_vars=(1.0, y0),
17        maximum_iterations=event_size)
18    return y
```

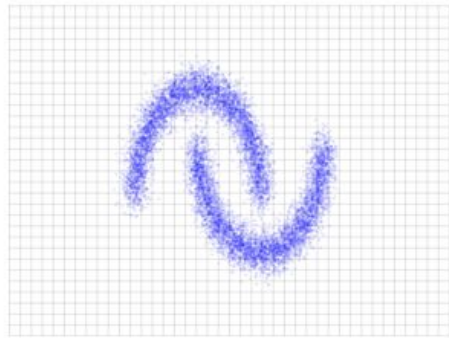

Normalizing flows - inference and generation

- Let us recall once again difference between inference and generation

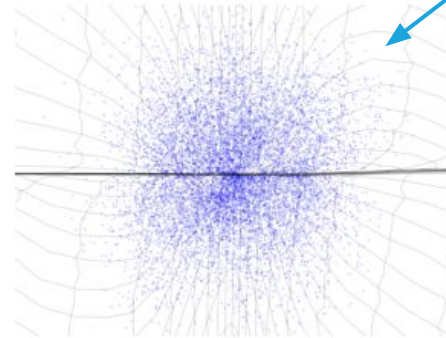
Inference

$$x \sim \hat{p}_X$$
$$z = f(x)$$

Data space \mathcal{X}



Latent space \mathcal{Z}



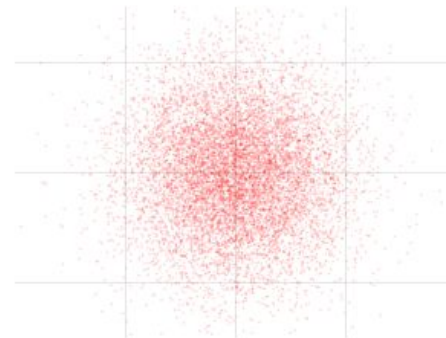
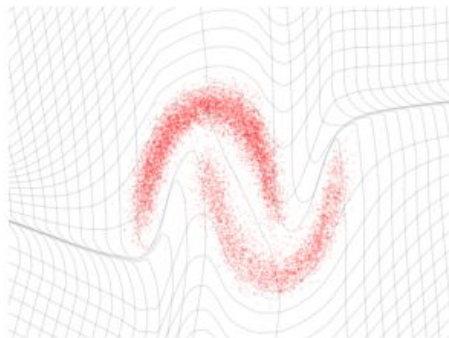
note the space distortion!

Used when we are interested in MLE:

- inverse transform
- inverse log det jacobian
- no reconstruction loss

Generation

$$z \sim p_Z$$
$$x = f^{-1}(z)$$



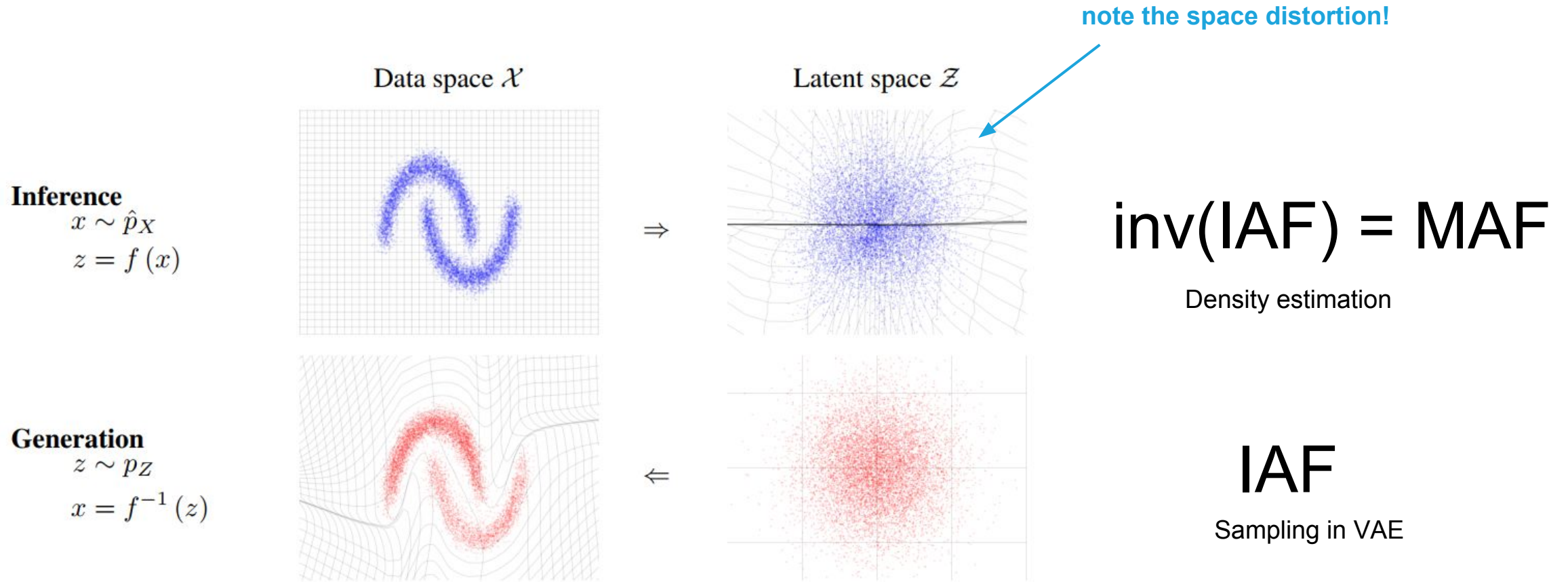
Used when we are interested in sampling: mostly related to VAEs

- forward transform
- forward log det jacobian, for estimation of density of model samples
- we need to add reconstruction loss

[Density estimation using Real NVP](#) (2016)

Normalizing flows - inference and generation

- Let us recall once again difference between inference and generation



[Density estimation using Real NVP](#) (2016)

Masked Autoregressive Flow for Density Estimation (**Edinburgh/2017**)

MAF: Basic idea

- Paper: [Masked Autoregressive Flow for Density Estimation](#)

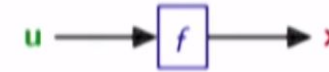
MAF is implemented as Bijector:

```
# A common choice for a normalizing flow is to use a Gaussian for the base
# distribution. (However, any continuous distribution would work.) E.g.,
maf = tfd.TransformedDistribution(
    distribution=tfd.Normal(loc=0., scale=1.),
    bijector=tfb.MaskedAutoregressiveFlow(
        shift_and_log_scale_fn=tfb.masked_autoregressive_default_template(
            hidden_layers=[512, 512])),
    event_shape=[dims])
```

IAF is an inverse of MAF:

```
# [Papamakarios et al. (2016)][3] also describe an Inverse Autoregressive
# Flow [(Kingma et al., 2016)][2]:
iaf = tfd.TransformedDistribution(
    distribution=tfd.Normal(loc=0., scale=1.),
    bijector=tfb.Invert(tfb.MaskedAutoregressiveFlow(
        shift_and_log_scale_fn=tfb.masked_autoregressive_default_template(
            hidden_layers=[512, 512]))),
    event_shape=[dims])
```

Masked Autoregressive Flow



- ✓ Fast to calculate $p(\mathbf{x})$
- ✗ Slow to sample from

Inverse Autoregressive Flow



- ✗ Slow to calculate $p(\mathbf{x})$
- ✓ Fast to sample from

MAF: Validating your model

- The model quality can be easily validated by projecting input images into random variables, they should follow the gaussian distribution

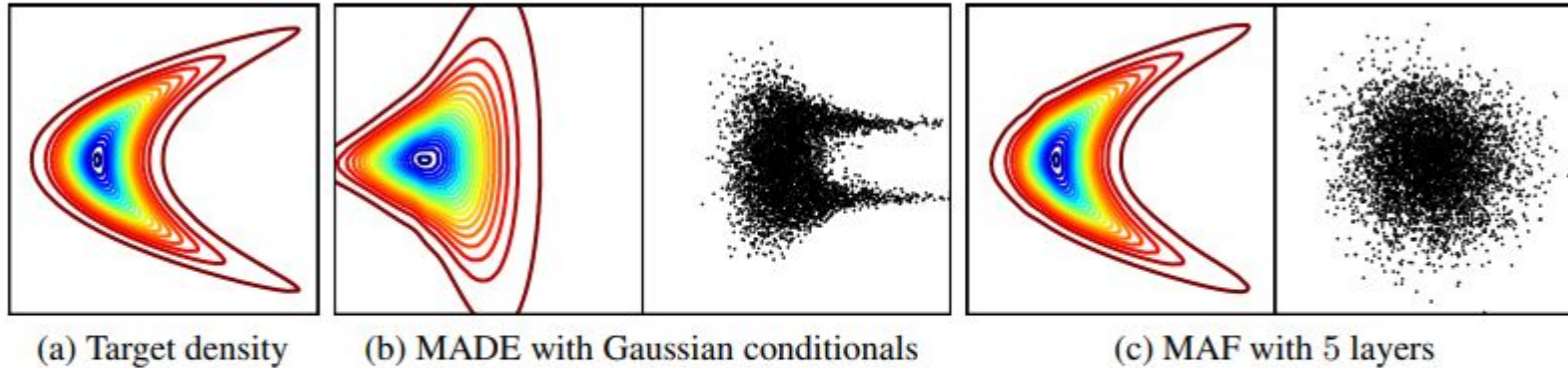


Figure 1: **(a)** The density to be learnt, defined as $p(x_1, x_2) = \mathcal{N}(x_2 | 0, 4)\mathcal{N}(x_1 | \frac{1}{4}x_2^2, 1)$. **(b)** The density learnt by a MADE with order (x_1, x_2) and Gaussian conditionals. Scatter plot shows the train data transformed into random numbers \mathbf{u} ; the non-Gaussian distribution indicates that the model is a poor fit. **(c)** Learnt density and transformed train data of a 5 layer MAF with the same order (x_1, x_2) .

Summary

Content (in past tense)

Part 1. Introduction

- Motivations for probabilistic models
- Theory of Normalizing Flows + simple examples
- Using Normalizing Flows in Tensorflow

Part 2. Research papers study

- MADE, IAF, MAF - require tricks, hacks and practice to train model

Content (in past tense)

Next time:

- Papers: NICE, RealNVP, Glow



Questions?



FORNAX

Introduction to Normalizing Flows

Krzysztof Kolasiński

WWW.FORNAX.AI

References:

- <http://www.machinelearning.ru/wiki/images/f/f2/LebedevTuzovaVaeNfSlides.pdf> - slides about Variational Auto-Encoder & Normalizing Flows
- https://docs.pymc.io/notebooks/normalizing_flows_overview.html - PyMC overview on NFs
- <https://github.com/ex4sperans/variational-inference-with-normalizing-flows> - some reimplementation of Reimplementation of Variational Inference with Normalizing Flows Paper
- <https://arxiv.org/pdf/1711.10604.pdf> - TensorFlow Distributions documentation
- <http://bjlkeng.github.io/posts/autoregressive-autoencoders/> - BLOG: very well written, with implementations and notebooks
- <http://ruishu.io/2018/05/19/change-of-variables/> - BLOG: CHANGE OF VARIABLES: A PRECURSOR TO NORMALIZING FLOW
- http://akosiorek.github.io/ml/2018/04/03/norm_flows.html - BLOG: Normalizing Flows
- <https://tonghehehe.com/blog/2017/1/1/vae> - Posterior Approximation for Variational Inference
- <https://blog.openai.com/glow/> - GLOW official website
- <https://towardsdatascience.com/autoregressive-models-in-tensorflow-96d69434cad9> - Blog: Autoregressive Models in TensorFlow

References:

- <http://ruishu.io/2018/03/14/vae/> - Blog: DENSITY ESTIMATION: VARIATIONAL AUTOENCODERS
- <https://vimeo.com/252105837> - Masked Autoregressive Flow for Density Estimation
- <https://arxiv.org/pdf/1705.07057.pdf> - MAF
- <https://arxiv.org/pdf/1502.03509.pdf> - MADE
- <https://arxiv.org/pdf/1606.04934.pdf> - IAF
- <https://arxiv.org/pdf/1505.05770.pdf> - Variational Inference with NFs
- <https://arxiv.org/pdf/1808.04730.pdf> - some recent paper



FORNAX

WWW.FORNAX.AI